一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2016-28(2016-08)

マルチグレイン並列化を支援する動画像処理環境の実装

古橋　一輝†　小野　和馬†　津邑　公暁†

† 名古屋工業大学

あらまし　静止画や動画像の処理を高速化するために SIMD 演算をサポートするプロセッサや複数のプロセッサコアを搭載するマルチコア環境が広く普及している．しかし，このような計算機環境の性能を引き出すためには，その実行環境に合わせたプログラムを記述する必要がある．そこで我々は，動画像処理プログラミングを抽象化することで，プログラマの負担を軽減する動画像処理ライブラリ RaVioli を提案している．しかし RaViol には，まだ動画像処理を十分に抽象化しきれていない部分があり，プロセッサの性能も引き出しきれていない．そこで本稿では，RaVioli よりも抽象度の高い言語と，その言語で記述したプログラムから様々な粒度での並列化を適用したアセンブリプログラムを自動生成するコンパイラの 2 つから成る動画像処理環境を提案する．この動画像処理環境を用いることで，プログラマは簡単な記述のみでプロセッサの性能を十分に引き出す動画像処理プログラムが開発可能となる．

キーワード　動画像処理ライブラリ，動画像処理環境，並列処理

1. はじめに

静止画や動画を扱える様々な情報機器の普及に伴い，マルチメディアアプリケーションを開発する機会が増加している．このようなアプリケーションの実行を高速化するために，単一の命令で複数のデータセットを同時に処理できる SIMD 演算をサポートするプロセッサが広く普及している．そして現在もなお，SIMD 演算のための命令セットアーキテクチャの拡張が続けられている．さらに，複数のプロセッサコアを搭載するマルチコア環境が広く普及している．しかし，このような計算機環境の性能を引き出せるよう動画像処理プログラムを開発するためには，その実行環境に合わせたプログラムを記述し，さらにこれをチューニングする必要がある．

この問題を解決するために，我々は解像度非依存型動画像処理ライブラリ **RaVioli**（**R**esolution-**A**daptable **V**ideo and **I**mage **O**perating **Li**brary）[1] を提案している．RaVioli は 1 フレーム内の画素数およびフレームレートをプログラマから隠蔽することで動画像処理プログラミングを抽象化し，プログラマの負担を軽減している．しかし，RaVioli には 2 つの問題点が存在する．RaVioli が動画像処理を十分に抽象化しきれていない点，およびプロセッサの性能を引き出しきれていない点である．

そこで本稿では，RaVioli よりも簡単な記述でプロセッサの性能を十分引き出すことのできる動画像処理環境を提案する．この動画像処理環境は大きく分けて，RaVioli よりも抽象度の高い言語と，その言語で記述したプログラムから様々な粒度での並列化を適用したアセンブリプログラムを自動生成するコンパイラの 2 つから成る．

2. 関連研究

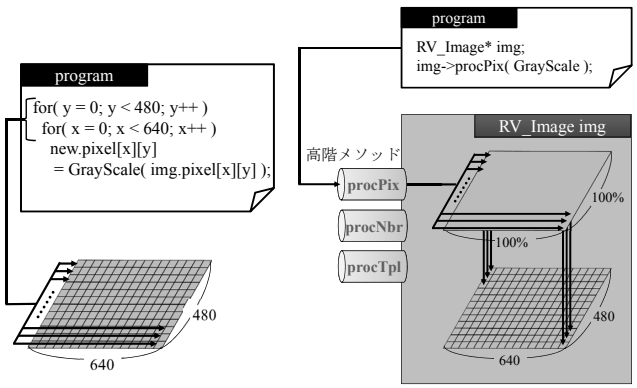
画像処理や動画像処理のためのライブラリとして OpenIP [2] や，OpenCV [3] などが提案されている．これらのライブラリは画像処理を実現するプログラムの記述を容易にする．また，OpenMP [4] などの並列化ライブラリと併用することで画像処理プログラムを並列化することが可能である．しかし並列化が有効な箇所を抽出するための知識や，並列化のための特有の記述の習得などが，プログラマに求められる．

また画像処理向けのプログラミング言語としては，金井らによる画像処理言語 [5] や Halide [6] などが提案されている．金井らは数式エディタを用いて記述できる独自の言語を提案し，画像処理プログラミングを抽象化している．この言語では処理単位となる画素配列の大きさを定義し，その配列の要素に対して処理を記述するため，ループ構造を用いないプログラミングが可能である．しかし，画像の解像度を意識した記述が必要であるなど，画像処理を十分抽象化しきれていない．Halide は汎用マルチコアプロセッサや GPU，モバイルプロセッサなど様々なプラットフォーム上での並列化を可能にする画像処理言語である．Halide を用いたプログラムでは画像処理アルゴリズム定義部分と並列化手順定義部分を分けて記述する．これによりプログラマは画像処理アルゴリズムに変更を加えることなく，最適な並列化スケジューリングを模索できる．さらに，プログラマが記述した画像処理アルゴリズムに，自動的に並列化スケジューリングを施す機能を備えている．しかし，動画像における複数フレーム間のスケジューリングに対しては，別途プログラマが行わなければならない．

これらに対し，本稿で提案する動画像処理環境では，簡単な記述のみで計算機環境の性能を引き出すことを可能にする．この動画像処理環境では，処理パターンに依存しない統一的な記述方式を採用するとともに，自動的に様々な粒度での並列化を適用したプログラムを生成するコンパイラを提供することで，プロセッサの性能を引き出す動画像処理プログラムの開発を容易にする．

3. 動画像処理ライブラリ RaVioli

一般に画像処理には，画像の構成要素に対する処理を，画像全体または任意の範囲に繰り返し適用するものが多い．例え



(a) 一般的なプログラム　　(b) RaVioli を用いたプログラム

図 1　画像処理プログラム

ば，カラー画像からモノクロ画像へ変換する処理では，処理単位は画素であり，その画素に対する単位処理が画像を構成する画素全てに適用される．一般的な画像処理プログラムの場合，**図 1(a)** に示すようにループイタレーションを用いることで全ての画素に対して処理が適用されるが，この際，プログラマは画像の幅と高さに対応した繰り返し回数を指示する必要がある．

一方 RaVioli では，画像の構成要素に対する処理のみを関数として定義し，その関数を RaVioli が提供しているメソッドに渡すことで，画像内の全構成要素に対して処理を適用することが可能である．RaVioli ではこの，構成要素に対する処理を記述した関数を**構成要素関数**と呼び，その構成要素関数を引数にとるメソッドを**高階メソッド**と呼ぶ．この高階メソッドは様々な処理単位ごとに用意されている．ここで，RaVioli を用いてカラー画像をグレースケールに変換する処理の様子を**図 1(b)** に示す．RaVioli は画像情報を RV_Image クラスにカプセル化している．このカプセル化された画像に対して処理を適用するには，RV_Image インスタンスの高階メソッドに構成要素関数を渡すのみでよい．図 1(b) の例では，RV_Image インスタンス img が持つ高階メソッド procPix() に，構成要素関数 GrayScale() を渡している．これにより procPix() は，img が持つ画像の全ての画素に，GrayScale() を順に適用する．このように RaVioli を用いる場合，プログラマは解像度を意識することなく画像処理プログラムを記述できる．また，RaVioli を用いて記述されたプログラムでは，画像の構成要素に対する処理が関数として切り出されるため，並列処理の単位が明確になり，自動並列化の実現が比較的容易である．そこで RaVioli では汎用マルチコアプロセッサや GPU，Cell/B.E. [7] など様々な環境に対応した自動並列化が実現されている [8]．しかし，RaVioli には大きく 2 つの問題点が存在する．1 つは，RaVioli が画像処理を十分に抽象化しきれていない点である．RaVioli を用いて画像処理プログラムを記述する際には，プログラマは目的とするアプリケーションの処理内容に適した高階メソッドを選択しなければならない．そして 2 つ目は，プロセッサの性能を十分に引き出しきれていない点である．RaVioli では，様々な環境に対応した自動並列化が実現されている．しかし，プロセッサの性能を引き出すために必要な様々な粒度での並列化には対応できていない．また，構成要素に対する処理が関数として定義された上で繰り返し適用されるという枠組みのため，関数呼び出しオーバヘッドが大きくなるという，性能に対する弊害もある．

```
1  (image)img1 > Grayscale > (image)img2{
2      (pixel)p1@img1{
3          ave = (p1.R + p1.G + p1.B) / 3;
4          p1.{R, G, B} = {ave, ave, ave};
5      }
6  }
7  (stream)st1 > StreamGray > st1{
8      (image)frame1@st1{
9          frame1 > Grayscale > frame1;
10     }
11 }
```

図 2　提案する言語で記述したグレースケール化プログラム

4. 提案する動画像処理環境

本章ではまず，本稿で提案する動画像処理環境の概要について述べる．その後，提案する動画像処理記述言語の仕様について述べ，最後に様々な粒度での並列化について述べる．

4.1 提案する動画像処理環境の概要

3 章で述べた問題点を踏まえて，より簡単な記述のみでプロセッサの性能を引き出すことができる動画像処理環境を提案する．提案する動画像処理環境では，抽象度の高い言語と，その言語で記述したプログラムから様々な粒度での並列化を適用したアセンブリプログラムを自動生成するコンパイラを提供する．まず提案する言語では，我々がこれまでに提案してきた画像処理向け言語 [9] の仕様を踏襲し，処理パターンに依存しない統一的な記述方式を採用する．これにより，プログラマは RaVioli よりも抽象度の高いプログラミングが可能となる．

一方，提案するコンパイラは，動画像処理に存在する 2 つの並列性に着目し，ベクトル化やブロック分割，パイプライン化といった様々な粒度での並列化を適用したアセンブリプログラムを生成する．これにより，並列処理や計算機環境の知識をプログラマが一切持たなくとも，性能を引き出すことを可能にする．

また，提案する言語では，構成要素に対する処理を関数として定義する考え方は従来の RaVioli を踏襲するが，C++コードへの変換時にコンパイラがこれをインライン化することで，RaVioli で生じていた関数呼び出しオーバヘッドも抑制できる．

4.2 高い抽象度を持つ動画像処理記述言語

提案する言語では RaVioli と同様に，画像や動画像の画素数およびフレームレートを意識する必要のないプログラミングパラダイムを提供する．さらに，処理単位と処理範囲を指定することでループ文を省略し，記述方式を大きく変更せずとも，様々な処理パターンを統一的に記述可能とする方式を採用する．

4.2.1 画像処理プログラムの記述

提案する言語を用いて記述したグレースケール化プログラムを**図 2** に示す．このプログラムは，1〜6 行目の画像に対する処理が定義された部分と 7〜11 行目の動画像に対する処理が定義された部分から構成されている．ここではまず，画像に対する処理が定義された 1〜6 行目を例に，提案する言語を用いて画像処理を記述する方法を説明する．まず，1 行目では関数名を定義している．また関数に対する入出力変数は，その関数名に対して “>” を介して隣接するように記述することで定義する．このプログラムでは，入力変数 img1 と出力変数 img2 をとる，Grayscale という名前の関数が定義されている．ここで，提案する言語では動画像処理プログラムの記述を容易にするために，いくつかの型を提供している．例えば，画像処理のため

```
1 (stream)in > main > (stream)out{
2     in > Binary | Edge > out;
3 }
```

図 3　main 関数の記述方法

の基本的な型として単一画素を表す pixel 型や部分画像を表す box 型を提供するほか，画像である 2 次元画素配列とは異なる用途向けの配列変数のために array 型なども提供している．なお，型が宣言されていない変数はスカラ変数として扱われる．各型の変数は，変数名の直前に “(” と “)” で囲って型名を表記することで宣言できる．図 2 の例の場合，1 行目において，入力変数 img1 は単一画像を意味する image 型で宣言されている．

ではここから，入力変数である img1 に対する処理内容が記述されている 2〜5 行目について説明する．まず，2 行目の “@” の左側は処理単位を右側は処理範囲を表している．すなわち “(pixel)p1@img1” は，以降で pixel 型の処理単位 p1 に対する処理が定義されていること，その p1 は img1 内の任意の要素を表すこと，そして定義されている p1 に対する処理が img1 を構成する処理単位全てに適用されることを示している．したがって 3，4 行目に記述された画素 p1 に対する処理は，img1 内の全画素要素に適用される．まず，3 行目では，p1 の R 値，G 値，B 値の平均値が計算され，その値が変数 ave に代入される．そして 4 行目では，ave の値が p1 の R 値，G 値，B 値にそれぞれ設定されることで画素 p1 がグレースケール化される．

4.2.2　動画像処理プログラムの記述

提案する言語を用いて動画像処理を記述する場合，4.2.1 項で述べたような画像処理のための構成要素関数に加えて，動画像処理のための関数を定義する．提案する言語では入力変数および出力変数として，動画像を表す stream 型の変数を用いることで動画像処理のための関数を定義できる．提案する言語を用いて動画像処理を記述する方法を，入力動画像に対してグレースケール化を施すプログラムである図 2 を例に説明する．まず 1〜6 行目で定義されている関数 Grayscale は 4.2.1 項で述べたように画像をグレースケール化する関数である．つぎに，7〜11 行目で定義されている関数 StreamGray は動画像処理のための関数であり，入力変数および出力変数に stream 型変数である st1 が指定されている．この関数内では，stream 型の変数 st1 に対する処理が定義されている．8 行目の “(image)frame1@st1” は処理範囲が st1，処理単位が image 型の frame1 であることを表しており，この frame1 は st1 内の任意の動画像フレームに対応する．9 行目の記述によって，1〜6 行目で定義された処理 Grayscale() が，st1 に含まれる全ての frame1 に適用される．このように提案する言語で記述した動画像処理プログラムでは，画像処理のための関数を定義し，その関数を動画像処理のための関数内の処理単位である image 型変数に適用することで，様々な処理を動画像に施すことが可能となる．以上のようにして，動画像処理の場合も画像処理と同一の記述方式でプログラムを記述することが可能である．

4.2.3　main 関数の記述

プログラム実行時に最初に呼び出される main 関数の記述方法を，**図 3** に示すプログラム例を用いて説明する．提案する言語で main 関数を定義するには図 3 の 1 行目のように関数名を “main” とする．また 1 行目では，main 関数で使用される入力変数と出力変数が，動画像処理のための関数と同様に stream 型で宣言されている．このとき入力変数および出力変数は，プログラム実行時の入力動画像および出力動画像と対応する．また，画像処理の場合では，入力変数および出力変数は image 型で宣言され，入力画像および出力画像と対応する．そして関数内では，プログラマ自身で定義した画像処理または動画像処理のための関数が入力変数に適用され，その結果が出力変数に代入される．ここで，一般的に画像処理および動画像処理には，複数の処理を段階的に適用し，最終的な結果を得るものが多く存在する．このような場合，提案する言語では図 3 の 2 行目のように処理を適用したい順に左から各関数を “|” で接続する形で記述することで段階的に適用する処理を実現できる．図 3 では，関数 Binary，Edge の 2 つがこの順に入力動画像 in の各フレームに適用され，結果として動画像 out が出力される．なお，“|” で接続された関数の出力は，次の関数の入力として渡される．このようにして，提案する言語は複数の処理から構成される動画像処理の記述を可能にし，様々な動画像処理に対応する．

4.3　様々な粒度での並列化

提案するコンパイラでは，前節で述べた言語で記述されたプログラムから様々な粒度での並列化を適用したアセンブリプログラムを自動生成する．ここで，提案手法では動画像処理に存在する 2 つの並列性に着目する．

まず 1 つ目は**データ並列性**である．一般的な画像処理において，1 画素や近傍画素集合などに対する処理がループ文を用いた繰り返し処理により画像全体に適用される場合その処理にはデータ並列性が存在する．このような場合，画像処理の内容に応じて入力画像の構成要素に対する処理にベクトル化を適用する．これを実現するために，複数のデータに対して同一の演算を同時に適用する SIMD 演算を用いる．これにより，ベクトル化しない場合と比較して複数のデータを単一の命令で処理することができ，実行する命令数を削減することが可能となる．

さらにデータ並列性が存在する場合，ベクトル化だけでなくブロック分割も有効である．提案手法では，入力画像を分割しその部分画像への処理を CPU の各プロセッサコアにそれぞれ割り当て，複数のプロセッサコアを同時に活用する．なお，このブロック分割は上述したベクトル化と組み合わせることが可能である．入力画像内の部分画像それぞれを複数のプロセッサコアが並列に処理し，さらに各プロセッサコア内でその部分画像の複数の画素を SIMD 演算を用いて並列に処理することで更なる高速化が期待できる．

そして，もう 1 つの並列性は**タスク並列性**である．一般的に動画像処理には，複数の処理を段階的に実行し，最終的な結果を得るものが多く存在する．中でも異なるフレームに対する複数の処理間にはタスク並列性が存在する．このような場合，動画像処理にパイプライン化を適用する．例えば，2 値化，エッジ抽出，ハフ変換，逆ハフ変換の 4 つから構成される動画像処理を実行する場合，これら 4 つの処理をそれぞれパイプラインステージとして扱い，各プロセッサコアに，各ステージの処理を**図 4** に示すように並列に実行させる．なお，この図における各処理の左上の数字は処理対象フレーム番号を表している．このパイプライン化もブロック分割と同様，ベクトル化と組み合わせることができる．以上で述べた様々な粒度での並列化を組み合わせることで，プロセッサの性能を十分に引き出すことが可能となる．

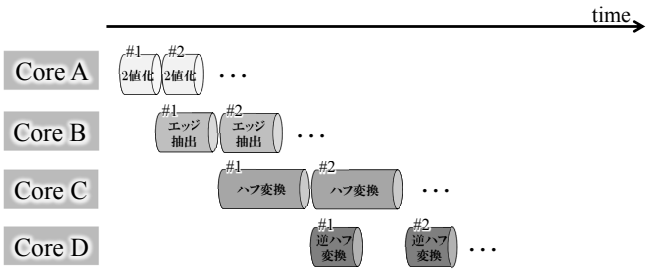


図 4　パイプライン実行による高速化

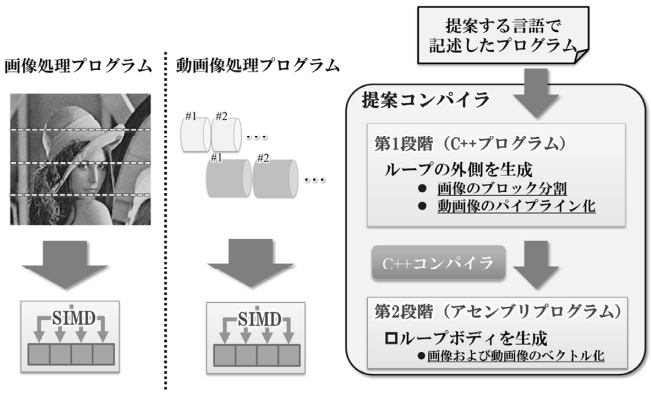


図 5　提案するコンパイラの概要図

5. コンパイラの実装

4.2 節で提案した言語で記述したプログラムから，4.3 節で述べた様々な粒度での並列化を適用したアセンブリプログラムを生成するコンパイラを実装した．本章ではまず，提案するコンパイラの概要を述べる．そして，提案するコンパイラがアセンブリプログラムを生成する流れを順に説明する．

5.1 コンパイラ概要

図 5 に示すように，提案するコンパイラは，様々な粒度での並列化を適用したアセンブリプログラムを 2 段階に分けて生成する．まず第 1 段階では，提案する言語を用いて記述したプログラムを入力として，画像のブロック分割および動画像のパイプライン化の部分，すなわち一般的な画像処理および動画像処理プログラムにおけるカーネルループの外側の処理のみを C++プログラムとして生成する．このブロック分割およびパイプライン化は Pthread を用いて実現する．そして，生成した C++プログラムを C++コンパイラでアセンブリ変換する．その後第 2 段階では，その出力に対して，画像・動画像処理プログラムにおけるカーネルループのループボディに相当する部分に，構成要素に対する処理に対応する命令を挿入する．ここで，提案するコンパイラではプログラムにベクトル化を適用するために，SIMD 専用命令をアセンブリプログラム中で挿入してベクトル化を適用する．以降の節では，提案するコンパイラが様々な粒度での並列化を適用したアセンブリプログラムを生成する流れを説明する．

5.2 ブロック分割プログラムの生成

まずコンパイラは，提案する言語で記述されたプログラムの main 関数を解析し，C++プログラムの main 関数を生成する．例として，**図 6** に示す main 関数がどのように生成されるかを説明する．コンパイラは記述されたプログラムの main 関数内の入力変数および出力変数から，画像の読み出しと書き込みの

```
1  int main(int argc,char* argv[]){
2     //入力画像の読み出し
3     Image* in = new Image();
4     IOHandler io;
5     io.Input(argv[1], in);
6     //プロセッサコア数のスレッドを生成
7     for(int i=0; i<THRNUM; i++){
8        if(pthread_create(&thread[i], NULL, Grayscale, (void*)in) != 0)
9           exit(1);
10    }
11    //スレッドの統合
12    for(int i=0; i<THRNUM; i++)
13       pthread_join(thread[i], NULL);
14    //出力画像の書き込み
15    io.Output(argv[2], in);
16    return 0;
17 }
```

図 6　第 1 段階を経て生成されるコード (main 関数)

```
1  void* Grayscale(void* image){
2     Image *img = static_cast<Image *>(image);
3     int width = img->getWidth();
4     int height = img->getHeight();
5     //スレッド ID の取得
6     int mythreadID = getmyThrID();
7     //処理担当範囲の計算
8     int sj = (height/THRNUM)*mythreadID;
9     int ej = sj+(height/THRNUM);
10    if(mythreadID == THRNUM-1)
11       ej = height;
12    //カーネルループ文
13    for(int j=sj; j<ej; j++){
14       for(int i=0; i<width; i+=CALC_WIDTH){
15          asm("nop");
16       }
17    }
18    return NULL;
19 }
```

図 7　第 1 段階を経て生成されるコード (関数 Grayscale)

ためのコードを生成する（3〜5 行目，15 行目）．その後，各プロセッサコアに割り当てるスレッドを生成するために，Pthread が提供しているスレッド生成関数の呼び出しを挿入する（7〜10 行目）．この時生成するスレッド数は，コンパイラがマシン情報をあらかじめプロファイルして取得したプロセッサコア数とする．なお，スレッド生成と同時に各スレッドで処理を担当させる関数をそれぞれ指定する．そして各プロセッサコアに割り当てたスレッドを関数の処理後に破棄するために，Pthread が提供しているスレッド破棄関数の呼び出しを挿入する（12〜13 行目）．

その後，コンパイラは構成要素関数を解析し，入力画像に対して処理を適用する関数を生成する．例として，**図 7** に示す構成要素関数 Grayscale がどのように生成されるかを説明する．まず，コンパイラは入力画像の幅と高さを取得するコードを生成する（3〜4 行目）．そして，入力画像の中でどの範囲を各スレッドが担当するかを算出するために，それぞれのスレッドの識別に必要となる ID を取得する関数呼び出しを生成する（6 行目）．なお，スレッド数が N のとき，この ID の値は 0 から $N-1$ のうちいずれかとなる．その後，この ID と，画像の幅や高さの値，そしてスレッド数から，入力画像の中でどの範囲を各スレッドが担当するかを算出する．この各スレッドが担当

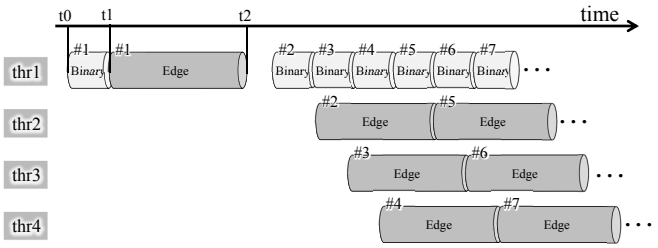


図 8　パイプライン実行の様子

表 1　評価環境

OS	CentOS 6.4
CPU	Intel Core i7-4770
Clocks	3.4 GHz
Memory	16 GB
Cores	8
SIMD Register Size	256 bits
Compiler	gcc 4.8
Compile options	-O3

する範囲は，空間的局所性を考慮し，メモリ上の連続した領域に対してアクセスできるように，画像を行方向に分割して区切られた範囲とする（8〜11 行目）．最後に，画像の各画素を処理するためのカーネルループ文を挿入する（13〜17 行目）．コンパイラは，このカーネルループ内に画像の構成要素に対する処理をコンパイルの第 2 段階で挿入していく．このときにコンパイラがそのループ文を検出できるよう，ループ文の中に nop 命令を挿入する（15 行目）．ここで，このカーネルループ文の再初期化式で使用している記号定数 CALC_WIDTH は，単一の SIMD 命令で処理されるデータ数，つまり並列幅を表しており，あらかじめプロファイルしたマシン情報から算出する．以上のように，画像処理プログラムの場合は，第 1 段階においてコンパイラは画像のブロック分割を実現するコードのみを生成する．

ただし，提案する言語を用いて記述したプログラム内に，各スレッドがアクセス競合を引き起こす可能性がある共有変数が存在する場合，その変数へのアクセスを排他制御する必要がある．提案する言語で記述したプログラムでは，ループを用いた従来の記述とは異なり，各構成要素の処理順が規定されていないため，共有変数へのアクセス制御は，**リダクション処理**を追加することにより実現可能である．そこでコンパイラは，共有変数を構成要素関数内で検出した場合，共有変数ごとにスレッドローカル変数を定義した上で，この変数に対してデータを読み書きしながら並列に処理を行うコード，および各スレッドが処理した結果を逐次的に統合するコードを生成する．コンパイラはこのようにしてリダクション処理を追加する．

5.3　パイプライン化プログラムの生成

動画像処理プログラムの場合，提案するコンパイラは第 1 段階としてパイプライン化を実現するコードを生成する．コンパイラはまず，ブロック分割プログラムのときと同様に提案する言語を用いて記述したプログラム内の main 関数および動画像処理のための構成要素関数を解析し，C++プログラムの main 関数を生成する．コンパイラは main 関数内の入力変数および出力変数から，動画像の読み出しと書き込みのコードをそれぞれ生成する．次にコンパイラは，記述された動画像処理において，各フレームに対する処理が複数の関数から構成されているか否かを検出する．もし，複数の関数から構成されている場合，それらの関数をパイプライン化によりオーバラップ実行できると判断し，関数の処理負荷比に基づいて各関数の適切な担当スレッド数を算出した上で，算出結果に基づき各関数に対してスレッド割り当てを行うコードを生成する．そしてコンパイラは main 関数を生成後，画像を処理する構成要素関数を解析し，入力動画像の各フレームに対して処理を適用する関数を生成する．

ここで，2 値化，エッジ抽出を順に行う動画像処理プログラムをパイプライン処理する際の動作の流れを，**図 8** に示す．まず，入力フレームの 1 フレーム目に対する処理のみを逐次実行し，各処理に要した時間を計測する（図 8，Binary（t0〜t1），Edge（t1〜t2））．この結果を用いて，もっとも処理の均衡化が見込めるように，各処理を担当するスレッド数を算出する．具体的には，各処理に要した実行時間の比を求め，その比から各処理を担当するスレッド数を算出する．例えば図 8 の場合，総スレッド数が 4 であり，Binary，Edge の実行時間比が 1 : 3 であるため，各処理を担当するスレッド数は，Binary が 1，Edge が 3 と算出される．その後スレッドを生成し，算出された数のスレッドを各処理に割り当て，2 フレーム目以降の処理をパイプライン実行する．

5.4　ベクトル化プログラムの生成

提案するコンパイラはコンパイルの第 2 段階としてベクトル化を実現するコードを挿入する．なお，第 2 段階では，提案する言語を用いて記述したプログラムと，第 1 段階を経て生成されたプログラムを C++コンパイラによって処理することにより得たアセンブリプログラムとを入力として，そのアセンブリプログラムにベクトル化を実現するコードを挿入していく．コンパイラはまず，第 1 段階を経て生成されたプログラムのアセンブリプログラムを解析し，カーネルループを検出するために，第 1 段階で挿入した nop 命令を探索する．nop 命令が挿入されているループ区間を検出したあと，コンパイラは，入力画像の画素データが格納されている画素配列の中から，処理に必要な画素データのアドレスを汎用レジスタへそれぞれ格納し，その画素データを SIMD 演算のための専用レジスタへ転送するための命令を挿入し，そのレジスタに格納された画素データを処理するための SIMD 専用命令を併せて挿入する．最後に，SIMD 演算用のレジスタ内の演算結果を画素配列に転送するための命令を挿入する．

6.　評　　価

提案手法の有用性を確かめるために，いくつかのサンプルプログラムを用いて評価を行った．本章では，提案する動画像処理環境で生成したプログラムの実行時間と既存の RaVioli 等を用いて実装したプログラムの実行時間を比較し，その結果を考察する．

6.1　画像処理の評価

まず，画像処理プログラムの実行時間を評価した．評価環境は**表 1** に示す通りである．評価には，グレースケール化，エッジ抽出，エンボスフィルタ，テンプレートマッチングの 4 つのプログラムを使用した．入力画像の解像度には一般的なスマートフォンで撮影可能な XGA（1024×768）を使用した．なお，テンプレートマッチングの際に用いるテンプレート画像には解像度が 120×120 である画像を使用した．

評価結果を**図 9** に示す．この図は各プログラムの実行時間を示している．図中では，各プログラムの評価結果が 3 本のバーで表されている．これらのバーは左から順に，C++で記述した

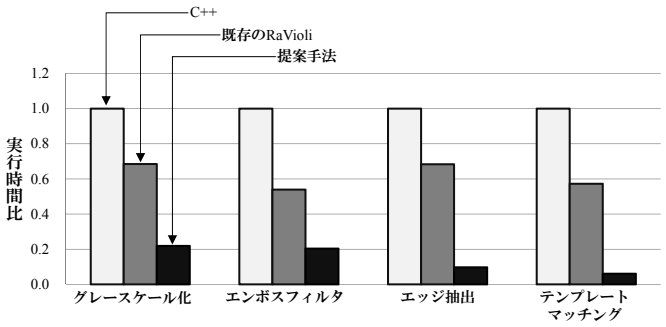


図 9　画像処理プログラムの実行時間

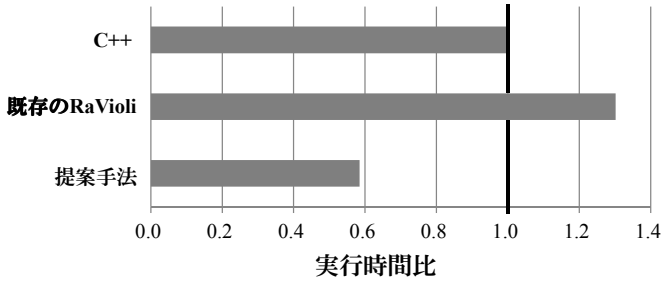


図 10　動画像処理プログラムの実行時間

プログラム，既存の RaVioli で記述したプログラム，そして提案する動画像処理環境で生成したプログラムの実行時間をそれぞれ表しており，C++プログラムの実行時間を 1 として正規化している．

評価の結果，提案する動画像処理環境で生成したプログラムは C++で記述したプログラムと比較して平均で 9.1 倍，最大で 16.4 倍速度が向上し，RaVioli で記述した並列プログラムと比較して平均で 5.5 倍，最大で 9.4 倍速度が向上した．このように提案する動画像処理環境で生成したプログラムでは既存の RaVioli と比較して，いずれのプログラムにおいても実行時間が短縮できていることが分かる．その中でもエッジ抽出プログラムとテンプレートマッチングプログラムで多くの実行時間が削減されている．これらはともに複数の構成要素関数を用いるプログラムであり，既存の RaVioli で実装した場合に関数呼び出しオーバヘッドの影響が大きい．提案手法では 4.1 節で述べたようにこのオーバヘッドが抑制できたため，これらのプログラムにおける既存の RaVioli に対する性能向上幅が大きくなったと考えられる．この結果から，提案する動画像処理環境によりプロセッサの性能をより引き出すことができることを確認した．

6.2　動画像処理の評価

次に，動画像処理プログラムの実行時間を評価した．評価には，2 値化，エッジ抽出，ハフ変換，逆ハフ変換の 4 つのステージで構成される直線検出プログラムを使用した．入力には，解像度が XGA で，長さが 30 秒間の動画像を使用した．なお，この動画像のフレームレートは 30fps であり，総フレーム数は 900 枚である．評価結果を**図 10** に示す．この図に示すバーは上から順に，C++で記述したプログラム，既存の RaVioli で記述したプログラム，提案する動画像処理環境で生成したプログラムのそれぞれの結果を表している．横軸は実行時間比を表しており，C++で記述したプログラムの実行時間を 1 として正規化している．なお，提案する言語で記述したプログラムでは，4.3 節で述べた通り，ブロック分割は適用せず，パイプライン化とベクトル化のみを適用する．また，パイプライン化に使用するスレッド数はコア数と同数の 8 スレッドとした．

評価の結果，提案する動画像処理環境で生成したプログラムでは，C++で記述したプログラムと比較して，1.7 倍速度が向上し，RaVioli で記述したプログラムと比較して 2.2 倍向上した．ここで本評価に用いた直線検出プログラムでは，ハフ変換の処理量が全体の 9 割以上を占めている．提案する動画像処理環境で各処理に割り当てられていたスレッド数を確認すると，ハフ変換に 5 スレッド，他の各処理にそれぞれ 1 スレッドずつ割り当てられていた．このようにハフ変換に多くのスレッドが割り当てられ，プロセッサ間の処理負荷が均衡化されていた．以上のように，使用するスレッド数を増やすことにより，各スレッドに割り当てられる処理負荷を均衡化させることができたため，速度が向上したと考えられる．

7.　おわりに

本稿ではまず，動画像処理ライブラリ RaVioli の問題点として動画像処理を十分に抽象化できていないこと，プロセッサの性能を十分に引き出しきれていないことを示した．これらの問題を解決するために，RaVioli よりも抽象度の高い動画像処理記述言語と，その言語から様々な粒度での並列化を適用したアセンブリコードを自動生成するコンパイラから成る動画像処理環境を提案した．提案する動画像処理環境の性能を確かめるために，いくつかの画像処理および動画像処理プログラムを用いて，C++および既存の RaVioli と実行速度を比較した結果，すべてのプログラムにおいて性能向上を確認し，C++で記述したプログラムに対して最大で 16.4 倍の速度向上を達成した．今後の課題として，動画像処理時に使用するスレッド数を自動調整する機能の実装が挙げられる．

文　献

[1] 岡田慎太郎他，“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.2，no.1，pp.63–74，Mar. 2009.

[2] G. Kovács, et al., “The openIP Open Source Image Processing Library,” Proc. Int’l Conf. on Multimedia (MM’10), pp.1489–1492, ACM, 2010.

[3] G. Bradski, et al., Learning OpenCV: Computer Vision With the OpenCV Library, O’Reilly & Associates Inc, 2008.

[4] L. Dagum, and R. Menon, “OpenMP: an Industry Standard API for Shared-Memory Programming,” IEEE Computational Science and Engineering, vol.5, 1998.

[5] J. Segawa, and T. Kanai, “The Array Processing Language and the Parallel Execution Method for Multicore Platforms,” The First International Symposium on Information and Computer Elements, 2007.

[6] J. Ragan-Kelley, et al., “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” PLDI ’13 Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, pp.519–530, ACM, 2013.

[7] Sony Computer Entertainment, Cell Broadband Engine Architecture, , 1.01 edition, Oct. 2006.

[8] 稲葉崇文他，“GPU 及び Cell/B.E. に対応した解像度非依存型動画像処理ライブラリ RaVioli の提案と実装，” 信学技報，vol.IEICE-110 (SWoPP2010)，no.IEICE-CPSY-167，pp.7–12 電子情報通信学会，Aug. 2010.

[9] A. Ono, et al., “A gpu-supported high-level programming language for image processing,” Proc. 7th Int’l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011), pp.245–252, Nov. 2011.