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
CPSY2017-133,DC2017-89（2018-03）

高抽象度言語とオートチューニング機能を持つ動画像処理環境

古橋　一輝†　　津邑　公暁†

† 名古屋工業大学

あらまし　複雑化する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，その実行環境の構成を考慮してチューニングする必要がある．また，実行環境によって最適なチューニングパラメータは大きく異なるため，チューニングされたプログラムは移植性が低くなってしまう．そこで本稿では，簡単な記述で複雑化する計算機の性能を引き出せる動画像処理環境を提案する．提案する動画像処理環境は高い抽象度を持つ専用言語とその言語からプログラムの特徴に応じた最適化が施されたプログラムへと自動変換する専用コンパイラから構成される．この動画像処理環境を用いて記述したプログラムを代表的な画像処理環境である Halide を用いて記述したプログラムと比較した結果，最大で約 6.9 倍実行時間が短くなった．

キーワード　動画像処理環境，オートチューニング，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，このような複雑化する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，その実行環境の SIMD レジスタのサイズ，キャッシュサイズ，コア数等を考慮してチューニングする必要があり，プログラム開発コストの大きさが問題となる．また，実行環境によって最適なチューニングパラメータは大きく異なるため，チューニングされたプログラムは移植性が低くなってしまう．

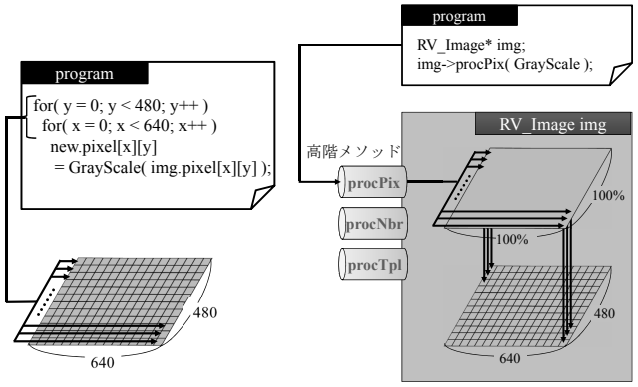
この問題を解決するために，簡単な記述で複雑化する計算機環境の性能を引き出すことを可能にする，動画像処理環境を提案する．提案する動画像処理環境は高い抽象度を持つ専用言語と，その言語から高性能なプログラムへと変換する専用コンパイラから構成される．専用言語では，処理パターンに依存しない統一的な記述方式を採用する．これにより，直観的で抽象度の高いプログラミングが可能となるだけでなく，並列処理可能な単位や，処理単位間の依存関係が明確となる．専用コンパイラは専用言語で記述されたプログラムからプログラムの特徴に応じたベクトル化，キャッシュブロッキング，マルチスレッド化が適用された高性能なプログラムへと自動変換する．

2. 関連研究

これまでに多くの画像処理や動画像処理のためのライブラリやフレームワークが提案されている．

2.1 RaVioli

我々は先に動画像処理ライブラリ RaVioli [1] を開発した．



(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

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．

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．

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

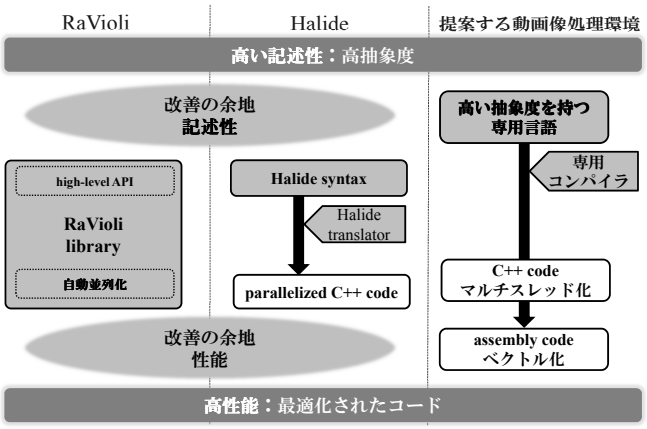


図 2　画像処理環境の概念図

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を関数として切り出して定義することになるため，並列処理可能な単位が明確になり，自動並列化が比較的容易であるため，様々な環境に対応した自動並列化が実現されている [2]．しかし，RaVioli には記述性と性能の 2 つの面で改善の余地が存在する．記述性の面では，プログラマは処理単位とその繰り返しパターンに応じて適切な高階メソッドを選択しなければならない．性能の面では，ベクトル化やマルチスレッド化といった様々な粒度での並列化を組み合わせることはできていない．

2.2　Halide

現在最も注目を集めている画像処理環境に Halide [3] がある．Halide は多数存在する他の画像処理環境と比較して，性能，可読性，記述性の面で高く評価されている [4]．そして，多くの研究に多大な影響を与え続けている [4–8]．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と並列スケジューリングを別々に記述する．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，自動的に並列スケジューリングを施す機能も備えている．しかし，図 2 に示すように Halide も RaVioli と同様，高い記述性と高性能とを十分に達成することができていない．例えば，画素の内容に応じて処理内容を変更する処理を条件文として記述する際に，組み込み関数を使用する必要があるなど，直感的なプログラミングが困難となる．また，Halide を用いて高性能なプログラムを生成するには，ハードウェアの構成を理解した上で，多数用意されている並列スケジューリングのための組み込み関数の中から最適な関数を選択し，さらに引数として，適切なパラメータを指定する必要があり，これは画像処理および，ハードウェアに精通していないプログラマにとって困難となる．この問題を解決するため，Halide の並列スケジューリングを自動で求めるオートチューナーとして Ansel らが提案している OpenTuner [9] や Mallapudi らが提案している Auto-scheduler [10] がある．OpenTuner を用いることで，シンプルなアルゴリズムであれば，1 時間程度で最適なスケジューリングを求めることが可能となる．しかし，複雑なアルゴリズムでは十分な最適化が得られない．これに対し，最新のオートチューナーである Auto-scheduler は OpenTuner よりも短い時間で Halide が用意している組み込み関数を用いて行うハンドチューニングとほぼ同等の性能を実現可能である．しかし，これらのオートチューナーを用いても高性能を実現しきることはできず，改善の余地が存在する．例えば，ベクトル化の並列幅を関数単位でしか指定できないといった Halide の言語仕様による制約が存在する．

```
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 3　専用言語で記述したグレースケール化プログラム

3.　提案する動画像処理環境

3.1　提案する動画像処理環境の概要

本稿では，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は，抽象度の高い言語と，その言語で記述されたプログラムから，プログラムの特徴に応じた最適化を施したプログラムを自動生成するコンパイラを提供する．まず専用言語では，我々がこれまでに提案してきた画像処理向け言語 [11] 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．そして専用コンパイラは，専用言語で記述されたプログラムにベクトル化，マルチスレッド化，およびキャッシュブロッキングを適用する．ここで，プログラムにはそれぞれ特徴があり，その特徴に応じた最適化をするためには，それぞれ固有のチューニングパラメータを設定する必要がある．そこで専用コンパイラは，プログラム毎にその特徴に応じた最適化を識別し，それが適用された高性能なプログラムへと自動変換する．

3.2　専 用 言 語

専用言語では 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．

3.2.1　画像処理プログラムの記述

専用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 3** 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，専用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，専用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 3 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．

3.2.2 動画像処理プログラムの記述

専用言語を用いて動画像処理を記述する場合，3.2.1 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．専用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．専用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 3 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 3.2.1 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．次に，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．このように専用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．

3.2.3 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 4** に示すプログラム例を用いて説明する．専用言語で main 関数を定義するには図 4 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，専用言語では図 4 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 4 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，専用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．

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

図 4　main 関数の記述方法

3.3 専用コンパイラ

専用コンパイラは，前節で述べた言語で記述されたプログラムにベクトル化，キャッシュブロッキング，マルチスレッド化を適用することで，高速な処理を実現する．まず専用コンパイラは，画像処理，動画像処理に存在する 2 つの並列性に着目してベクトル化，マルチスレッド化を適用する．まず 1 つ目の並列性は**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入力画像を複数の領域に分割し，その部分画像に対する処理を複数のスレッドに割り当て並列に処理させるマルチスレッド化を適用できる．なお，このマルチスレッド化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，動画像に対して適用する複数の処理をそれぞれパイプラインステージとして扱い，マルチスレッドによりパイプライン実行させることで高速化が期待できる．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別のスレッドで**図 5** 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．また，この

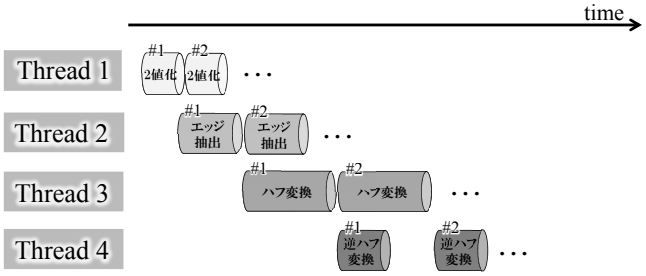


図 5　パイプライン実行による高速化

場合においてもベクトル化と組み合わせることができ，これにより更なる高速化が期待できる．

さらにキャッシュ利用効率を向上させるため，専用コンパイラはキャッシュブロッキング [12] を適用する．キャッシュブロッキングとは，キャッシュメモリの容量を考慮してループの構成を変更することで，キャッシュヒット率を向上させる手法である．フィルタ処理に代表されるように画像処理には，一度参照した画素データを繰り返し参照する処理が多く存在する．このような処理において，キャッシュブロッキングを適用していないプログラムでは，一度参照した画素データを再度参照する際，他の画素データへの参照によりキャッシュメモリから追い出されてしまう可能性がある．これに対し，キャッシュブロッキングを適用した場合，複数回参照するデータをキャッシュメモリから追い出される前に再び参照することができ，キャッシュ利用効率を向上させることで，処理の高速化が期待できる．

ここで，プログラムには，含まれる演算や参照する近傍画素の範囲が異なるなど，それぞれ特徴があり，その特徴に応じた最適化を施すことが望ましい．そこで，専用コンパイラは記述されたプログラムを解析し，プログラムの特徴からそのプログラムに適したベクトル化，キャッシュブロッキング，マルチスレッド化のためのチューニングパラメータを導き出す．そして，これにより得られた最適化を適用する．

4. オートチューニング機能

本章では，オートチューニング機能の実現に向けて行ったベクトル化，キャッシュブロッキング，マルチスレッド化に対する調査結果を示す．

4.1 ベクトル化

プログラムには必要となるデータ長が異なる様々な演算が含まれており，常に最大の並列幅で実行するためには，演算毎に並列幅を変更する必要がある．しかし，Halide はベクトル化を適用するにあたって，並列幅を関数単位でしか変更できない．これに対し，提案する動画像処理環境では演算単位で並列幅を変更できるように実装し，可能な限り最大の並列幅での実行を可能とする．ここで，並列幅を変更する際，それに伴うオーバヘッドが発生してしまうため，最大の並列幅での実行が必ずしも適切とは限らない．そこで，ベクトル化を適用するにあたって並列幅を変更すべきとなる変更基準を調査した．調査環境として，CPU は SIMD レジスタのサイズが 512bits である Intel Core i7-7900X，メモリは 64GB を搭載した PC を用い，OS には Ubuntu 16.04 を用いた．また，コンパイラには gcc 5.4.0 を用い，コンパイラオプションは-O3 を指定した．調査プログラムにはグレースケール化とブラーを用いた．どちらのプログラムもメモリから SIMD レジスタへのロード，乗算，加算，SIMD

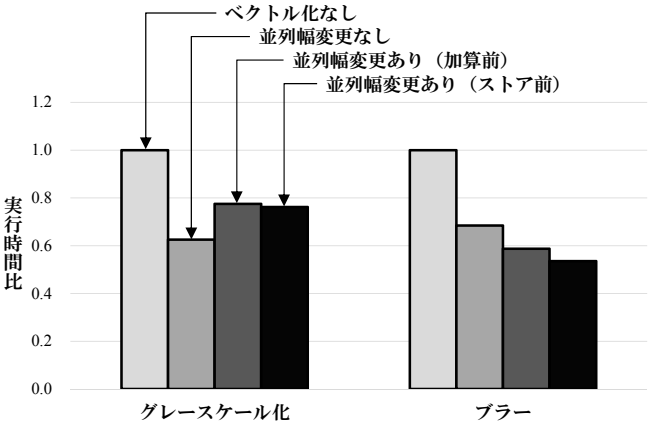


図 6　調査結果:ベクトル化

レジスタからメモリへのストアの順にこれらの処理がループにより繰り返し実行されていく．そして，SIMD 命令により同時に処理される各データにはそれぞれロードと乗算では 16bit，加算とストアでは 8bit の領域が必要となる．

調査結果を**図 6** 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，各プログラムの調査結果が 4 本のバーで表されている．これらのバーは左から順にベクトル化を適用していないプログラム，ベクトル化を適用し途中で並列幅を変更しないプログラム，並列幅を加算の前で変更するプログラム，並列幅をストアの前で変更するプログラムの実行時間を表しており，ベクトル化を適用していないプログラムの実行時間を 1 として正規化している．

まず，グレースケール化とブラーとを比較すると，グレースケール化では，並列幅を変更しないプログラムの方が実行時間が短くなったのに対し，ブラーでは，並列幅を変更したプログラムの方が実行時間が短くなった．グレースケール化における並列幅を変更しないプログラムでは，入力画像データが配置されているメモリ上の領域に出力を上書きする形をとっているが，並列幅の変更により入出力データの形式が変わり，出力領域を入力領域とは別に確保する形に変更する必要があり，これがキャッシュミスの増大を招いたため性能が悪化した．これに対し，ブラーにおける並列幅を変更しないプログラムは，入力画像データとは異なる領域に出力を行っている．そのため，並列幅を変更したことでストア回数の削減，ひいてはキャッシュミスの低減につながり，実行時間を短縮できた．次に，並列幅を加算の前で変更する場合とストアの前で変更する場合を比較すると，並列幅をストアの前で変更する場合の方が実行時間が短くなった．これは，並列幅の変更に伴うオーバヘッドが並列幅の変更による加算の削減時間よりも大きいためである．

以上より，並列幅を変更すべきかどうかは並列幅の変更に伴うキャッシュミスの変化に基づいて決めるべきであるということがわかった．また，並列幅を変更する適切なタイミングは並列幅変更のオーバヘッドと並列幅変更によって得られるレイテンシ削減との大小によって決めるべきであることがわかった．

4.2 キャッシュブロッキング

キャッシュブロッキングが有効であるには，一度参照した画素データをキャッシュメモリから追い出された後に，再度参照するという特徴が必要となる．そこで，どのようなプログラムにこの特徴があり，キャッシュミスを削減できるかを調査した．調査環境は 4.1 節で示した調査環境と同一であり，キャッシュ

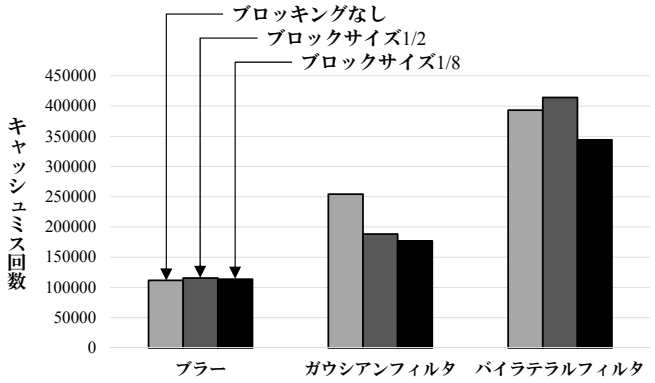


図 7　調査結果:キャッシュブロッキング

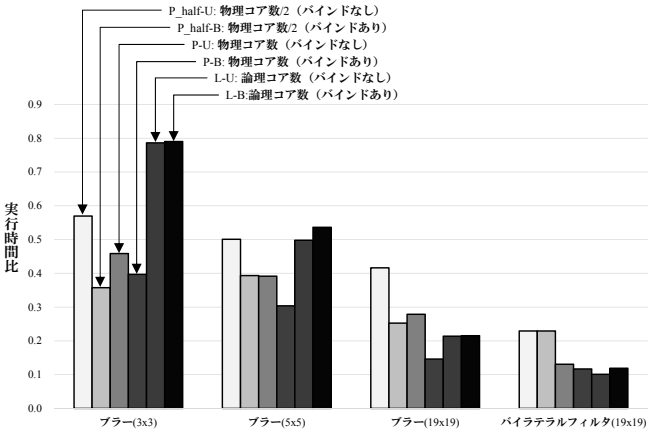


図 8　調査結果:マルチスレッド化

サイズは L1 d-cache が 32KB，L2 cache が 1024KB，LLC が 13.75MB である．調査プログラムにはブラー，ガウシアンフィルタ，バイラテラルフィルタを用いた．

調査結果を**図 7** に示す．この図は各プログラムのキャッシュミス回数を示している．図中では，各プログラムの調査結果が 3 本のバーで表されている．これらのバーは左から順にキャッシュブロッキングを適用していないプログラム，ブロックサイズを L1 d-cache の 1/2，1/8 としキャッシュブロッキングを適用したプログラムのキャッシュミス回数を表している．

まず，ガウシアンフィルタとバイラテラルフィルタではキャッシュミス回数の削減を確認できた．ガウシアンフィルタは 2 枚のフィルタを順にかける処理であり，1 枚目のフィルタ処理で参照した画素データを 2 枚目のフィルタ処理時に再度参照する．キャッシュブロッキングを適用していない場合では，キャッシュメモリから追い出された後に再度参照してしまう．これに対し，キャッシュブロッキングを適用し，ブロック毎に 1 枚目，2 枚目のフィルタを順にかけていくことで，キャッシュメモリから追い出される前に再度参照することができ，キャッシュミス回数を削減できたと考えられる．バイラテラルフィルタは非常に重たい演算である exp 計算が含まれている．この exp 計算の実行に時間を要してしまうため，その間にキャッシュメモリに格納されているデータが次々と更新されてしまう．そのため，キャッシュブロッキングを適用し処理の順番を工夫することで，キャッシュメモリから追い出される前に再度参照することができ，キャッシュミス回数を削減できたと考えられる．

次に，ガウシアンフィルタとバイラテラルフィルタにおけるブロックサイズを L1 d-cache の 1/2 と 1/8 にしたときを比較すると，どちらも 1/8 にしたときの方がキャッシュミス回数を削減できた．これは，ブロックサイズを L1 d-cache の 1/2 にしたときでは，キャッシュメモリから追い出される前に再度参照しきることができないかったことが原因だと考えられる．

以上より，キャッシュブロッキングを適用するかは複数の構成要素関数から構成されているかおよび，exp 計算のような重たい処理が含まれているかに基づいて決めるべきであることがわかった．また，適切なブロックサイズは L1 d-cache の 1/8 であることがわかった．

4.3　マルチスレッド化

スレッド数を増やすと並列度を上げることができるが，生成・統合に伴うオーバヘッドが増大してしまうため，プログラムに応じて適切なスレッド数は異なると考えられる．また，スレッドをコアにバインドすることで，コア毎に保有されているキャッシュを有効活用できる．しかし，RaVioli や Halide はスレッドをコアにバインドする機能を提供していない．これに対し，提案する動画像処理環境では，スレッドをコアにバインドできるように実装する．ここで，スレッドをコアにバインドした場合，コア間での負荷分散ができなくなってしまう．そこで，適切なスレッド数および，コアにバインドさせる基準について調査した．調査環境は 4.1 節で示したものと同一であり，CPU の物理コア数は 10，論理コア数は 20 である．調査プログラムにはブラーのフィルタサイズを 3×3，5×5，19×19 にした場合とフィルタサイズ 19×19 のバイラテラルフィルタを用いた．

調査結果を**図 8** に示す．図中では，各プログラムの実行に要した時間が 6 本のバーで表されている．これらのバーは左から順に物理コア数の半数のスレッドを用い，コアにバインドしない場合（P_half-U）とそれぞれ別コアにバインドした場合（P_half-B），物理コア数と同数のスレッドを用い，コアにバインドしない場合（P-U）とそれぞれ別コアにバインドした場合（P-B），論理コア数と同数のスレッドを用い，コアにバインドしない場合（L-U）と各物理コアに 2 スレッドずつバインドした場合（L-B）の実行時間をそれぞれ表しており，単一のスレッドを用い，コアにバインドしない場合の実行時間を 1 として正規化している．

まず，フィルタサイズを 3×3 としたブラーでは P_half-B，フィルタサイズを 5×5 と 19×19 としたブラーでは P-B バイラテラルフィルタでは L-U の実行時間が一番短くなった．ブラーでは，フィルタのサイズが大きくなるほど参照する画素が増え，処理量が増大する．さらに，バイラテラルフィルタは実行に多くの時間を要する exp 計算が含まれており，フィルタサイズが同じであっても，ブラーよりも処理量が多くなる．このことから，処理量が多くなるほど，多くのスレッドを用いた方が性能が向上することがわかった．

また，論理コア数と同数のスレッドを用いる場合は，スレッドをコアにバインドしない方が良く，それ以外の場合は，スレッドをコアにバインドした方が良いことがわかった．実際の計算機環境では，OS 等他のプロセスも動作しているため，調査プログラムによりされたスレッドを各コアが実行できない時間が存在する．L-B では，論理コア数と同数のスレッドを用いるため，常にこの時間が存在する．さらに各スレッドはバインドされたコアでしか実行できないため，あるコアで実行した調査プログラム以外の処理に要した時間の分だけ，そのコアに割り当てられるスレッドの実行に要する実行時間が増加してしま

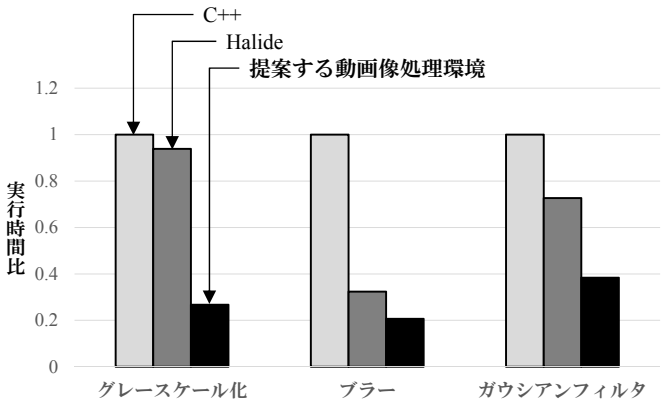


図 9　サンプルプログラムの実行時間

い，全体の実行時間も増加してしまう．そのため，L-B よりも L-U の実行時間が短くなったと考えられる．

以上から，フィルタサイズが大きくなるほど処理量が多くなると考え，フィルタサイズが 3×3 以下の場合，物理コア数の半数のスレッドを用い，フィルタサイズが 5×5 以上の場合，物理コア数と同数のスレッドを用い，それぞれ別コアにバインドすべきであることがわかった．さらに，exp 計算のような超越関数を含んでいる場合，論理コア数と同数のスレッドをコアにバインドせず用いれば良いことがわかった．

5. 評　　価

4 章に示したチューニング指針に基づき，プログラムに応じたベクトル化，キャッシュブロッキング，マルチスレッド化を施す専用コンパイラの有用性を確かめるために，いくつかのプログラムを用いて評価を行った．評価環境は 4 章で示した調査環境と同一である．評価には，グレースケール化，ブラー，ガウシアンフィルタの 3 つのプログラムを使用した．また，入力画像の解像度には QUXGA（3840×2400）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++で記述したプログラム，Halide 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．なお，Halide の並列スケジューリングは OpenTuner を用いて求めた．本評価に用いたプログラムでは，このチューニングに約 1 時間から 5 時間もの時間を要した．

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 Halide で記述したプログラムと比較して平均で 2.3 倍，最大で 6.9 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 Halide 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．これは，4. 章で述べた並列幅の変更やコアバインドといった，より綿密なチューニングを施すことができたためであると考えられる．この結果から，提案する動画像処理環境では Halide より適切なチューニングができることを確認した．

6. お わ り に

本稿では簡単な記述で複雑化する動画像処理環境の性能を引き出すことを可能とする動画像処理環境を提案した．提案する動画像処理環境は，高い抽象度を持つ専用言語と，その言語で記述されたプログラムにその特徴に応じた最適化を施す専用コンパイラから成る．この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プログラムを用いて，C++および Halide 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実行時間が最短となり，Halide で記述したプログラムに対して最大で 6.9 倍の速度向上を達成した．今後の課題として，適切なスレッド数を求めるためのより正確な処理量の測定法の考案が挙げられる．

謝辞　本研究の一部は，JSPS 科研費 17H01764 の助成による．

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，March 2009.

[2] 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.

[3] J. Ragan-Kelley, et al., “Decoupling algorithms from schedules for easy optimization of image processing pipelines,” ACM Transactions on Graphics (TOG) - SIGGRAPH 2012 Conference Proceedings, ACM, July 2012.

[4] R. Stewart, “An image processing language: External and shallow/deep embeddings,” Proc. 1st Int’l Workshop on Real World Domain Specific Languages (RWDSL’16), ACM, 2016.

[5] J. Hegarty, et al., “Rigel: flexible multi-rate image processing hardware,” ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016, ACM, July 2016.

[6] V. Korhonen, et al., “Rapid customization of image processors using halide,” Proc. IEEE Global Conference on Signal and Information Processing (GlobalSIP), pp.27–29, IEEE, 2014.

[7] J. Hegarty, et al., “Darkroom: compiling high-level image processing code into hardware pipelines,” ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014, ACM, July 2014.

[8] R.T. Mullapudi, et al., “Polymage: Automatic optimization for image processing pipelines,” Proc. 20th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’15), pp.429–443, ACM, 2014.

[9] J. Ansel, et al., “Opentuner: An extensible framework for program autotuning,” In Proceedings of the 23rd international conference on Parallel architectures and compilation, pp.303-316, ACM, August 2014.

[10] R.T. Mullapudi, et al., “Automatically scheduling halide image processing pipelines,” ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016, ACM, 2016.

[11] 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.

[12] M.D. Lam, et al., “The cache performance and optimizations of blocked algorithms,” ACM SIGARCH Computer Architecture News, pp.63-74, ACM, 1991.