Vol.2019-ARC-238 No.9
2019/11/15

記述性と性能を両立する動画像処理環境の検討と実装

高岡 昌弘[1]　津邑 公暁[1]

概要：高性能な動画像処理プログラムを開発するためには，プラットフォームに合わせたチューニングが必要である．しかし，この作業には計算機アーキテクチャや並列処理に関する高い知識が要求されるだけでなく，膨大な開発期間を要するため，プログラム開発コストの大きさが問題となる．そこで我々は，動画像処理プログラムを簡潔に記述できる専用言語およびその言語を用いて記述されたプログラムに対して，高度な最適化を施す専用コンパイラから成る動画像処理環境を提案している．本研究ではこの動画像処理環境に対し，従来よりも更に多くの処理を記述できるようにするため，専用言語を拡張した．また専用コンパイラに自動最適化機能を実装した．この改良した動画像処理環境の有用性を評価した結果，専用コンパイラが最適化したプログラムは，代表的な画像処理環境である Halide およびその Auto-scheduler を用いて最適化したプログラムより最大約 3.0 倍高速であることを確認した．

1. はじめに

画像処理技術が発展し，多分野で応用されている．これに伴い，動画像処理プログラムを開発する機会が増加している．高性能な動画像処理プログラムを開発するためには，プラットフォームに合わせてプログラムをチューニングする必要がある．しかし，この作業には，計算機アーキテクチャや並列処理に関する高い知識が要求される．それに加え，並列処理の際のスレッド数など，最適なパラメータを求めるためには，様々なチューニングを施したプログラムを実装し，それらの性能を検証する必要がある．そのため，高度なプログラミング技法と膨大な開発期間を要し，プログラマに負担を強いる．また最適なチューニングパラメータは，プラットフォームごとに異なるため，チューニングされたプログラムは，移植性が低下してしまう．これらの要因からプログラム開発コストの大きさが問題となる．

そこで我々は，簡潔な記述でプラットフォームが持つ性能を最大限に引き出す動画像処理環境 [1] を提案している．この動画像処理環境は，抽象度の高い専用言語とその言語を用いて記述されたプログラムに対して高度な最適化を施す専用コンパイラから構成される．これまでの研究では，専用言語の提案と専用コンパイラが施す最適化の適切なチューニングパラメータの調査を行ってきた．

本研究では，これまでの調査結果に基づき，プログラムに高度な最適化を施す専用コンパイラを実装した．また専用言語の拡張も行い，従来よりも更に多くのプログラムを記述することが可能となった．本稿では，専用コンパイラが最適化したプログラムの性能を評価し，我々が提案している動画像処理環境の有用性を実証する．

[1] 名古屋工業大学
Nagoya Institute of Technology

2. 先行研究

前章で述べたような問題を解決するために，画像処理や動画像処理プログラムの開発に特化した様々なフレームワーク [2–7] が提案されている．本章では，中でも記述性と性能の両方に優れたフレームワークである Halide および我々が提案しているフレームワークの概要について述べる．

2.1 Halide

現在最も注目を集めている画像処理環境に Halide [8] がある．Halide は，画像処理や配列処理を簡潔に記述できるように設計されたドメイン固有のプログラミング言語 (**Domain Specific Language: DSL**) であり，C++に組み込む関数型の言語として実装されている．また Halide は，マルチコア CPU，GPU，およびモバイルプロセッサなどの様々なプラットフォームに対応している．Halide が持つ最大の特徴は，プログラムを記述する際に，画像処理の「アルゴリズム」と，計算順序やデータアライメントに関する「スケジュール」とを分離した形で記述できることである．これにより，アルゴリズムを変更することなく，様々なスケジュールを施したプログラムが記述できる．そのため，Halide を用いて記述されたプログラムは，一般的なプログラミング言語を用いて記述されたプログラムと比べて，可読性が比較的高い．また Halide が提供するスケジュー

ル用の組み込み関数を用いることで，プログラマは様々なスケジュールを簡潔に記述できる．そのため，最適なスケジュールを求める際に要する時間を大幅に短縮することができる．しかし，Halide を用いても最適なスケジュールを指定する作業は容易ではない．最適なスケジュールを求めるためには，計算機アーキテクチャおよび種々の最適化を理解した上で，Halide が提供する多数の組み込み関数の中から適切な関数を選択し，最適なパラメータを指定する必要がある．この作業は，画像処理や計算機アーキテクチャに精通していないプログラマにとって困難である．そのため，Halide ではプログラマが記述したアルゴリズムに対して，自動的に適切なスケジュールを施す Auto-scheduler [9] を提供している．これを用いることで，プログラマはアルゴリズムを記述するだけで，Auto-scheduler が適切なスケジュールを推定し，プログラムに最適化を施すため，ある程度高性能なプログラムを生成できる．しかし，現状 Auto-scheduler は全てのアルゴリズムに対して最適なスケジュールを施すことはできない [9]．また Halide で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ため，各コアが保有するキャッシュを有効活用できず，複数スレッドで並列処理しても性能があまり向上しない可能性がある．

2.2 我々が提案しているフレームワーク

我々が提案しているフレームワークは，1 章で述べたように画像処理や動画像処理の記述に特化した専用言語と，その言語を用いて記述されたプログラムに対して，高度な最適化を施す専用コンパイラから構成される．専用言語は，抽象度の高い記述方式を採用したドメイン固有のプログラミング言語である．これは，我々がこれまでに提案してきた画像処理向け言語 [10] の仕様を踏襲している．本節では，その言語の特徴について述べる．

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，カラー画像をモノクロ画像へと変換する処理では，処理単位は画素であり，各画素に対する処理を全ての画素に対して適用する．一般的なプログラミング言語を用いる場合は，**図 1** に示すようにループ文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，専用言語では，ループ文の代わりに，処理単位と処理範囲を指定する．この記述方式を用いて図 1 と同様の処理を記述した例を**図 2** に示す．1 行目の “(pixel)p1” が処理単位を表し，“(image)img1” が処理範囲を表す．このように専用言語では，“処理単位@処理範囲” という形式でループ文を代替する．また専用言語は，画像処理や動画像処理プログラムの記述を容易にするために，複数のデータ型を提供している．専用言語が提供するデータ型の種類を**表 1** に示す．例えば，図 2 中 1 行目に用いられている pixel 型は単一画素を表し，image 型は単一画像を表す．したがって図 2 中 1 行目の “(pixel)p1@(image)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．

この記述方式には 3 つの利点が存在する．まず 1 つ目の利点として，記述方式を大きく変更することなく，様々な処理パターンを記述できることが挙げられる．ここで，図 2 に示した画像処理を動画像の各フレームに対して適用した例を**図 3** に示す．このプログラムは画像に対する手続き (1〜6 行目) と，動画像に対する手続き (7〜11 行目) から構成されている．専用言語では，関数に対する入出力変数を関数名に対して “>” を介して隣接するように記述することで定義する．専用言語を用いて動画像処理プログラム

```
1  for(y=0; y<480; y++){
2      for(x=0; x<640; x++){
3          ave=(img[x][y].R+img[x][y].G+img[x][y].B)/3;
4          img[x][y].R=img[x][y].G=img[x][y].B=ave;
5      }
6  }
```

図 1 一般的なプログラミング言語で記述したプログラム

```
1  (pixel)p1@(image)img1{
2      ave=(p1.R + p1.G + p1.B)/3;
3      p1.{R, G, B}={ave, ave, ave};
4  }
```

図 2 専用言語で記述したプログラム

表 1 専用言語が提供するデータ型の種類

型名	型が表す対象
pixel	単一画素
box	部分画像
image	画像
stream	動画像
array	配列 (ただし画素配列とは異なる要素向け)

```
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動画像処理プログラム

を記述する場合は，まず各フレームに対する画像を処理するための関数を定義し，動画像を処理する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できる．このように，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可能である．次に 2 つ目の利点として，画像処理および動画像処理プログラミングを抽象化できることが挙げられる．一般的なプログラミング言語を用いて画像処理プログラムを記述する場合は，図 1 中 1，2 行目のように画像の高さと幅を指定する必要がある．加えて動画像処理の場合は，フレームレートを考慮してプログラムを記述する必要がある．一方，専用言語を用いる場合は，それらの情報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．これにより，直感的で抽象度の高いプログラミングが可能となるため，プログラマの負担を軽減することができる．最後に 3 つ目の利点として，並列化可能な単位や処理間の依存関係が明確になることが挙げられる．図 1 のようにループ文を用いた場合は，イテレータによりループ内部の処理順序が固定されてしまう．しかし，図 1 に示すようなループ内部の処理順序によって処理結果が変わらない場合に処理順序が固定されると，コンパイラが並列化可能な単位の特定が困難になる場合が多い．一方，専用言語の記述方式を用いる場合は，任意の順序で実行可能な処理に対して，処理順序を固定することなく，それらの処理を記述できる．その結果，処理間の依存関係が明確になり，専用コンパイラが並列化可能な単位の抽出を容易にすることができる．

3. フレームワークの拡張

本章では，専用言語の拡張および専用コンパイラに実装した自動最適化機能について述べる．

3.1 専用言語の拡張

画像処理のブラーフィルタ，ガウシアンフィルタ，およびラプラシアンフィルタなどのフィルタ計算は，2 次元の畳込み演算を 1 次元のフィルタに分離して計算することができるため，計算コストの削減が可能である．このように分離して計算可能なフィルタは**セパラブルフィルタ**と呼ばれる．これまでの専用コンパイラでは，セパラブルフィルタの記述から実行コードを生成することができなかった．そこで本節では，専用コンパイラがセパラブルフィルタを処理できるように行った拡張について述べる．

まず，フィルタサイズ 3×3 のブラー処理を例に，セパラブルフィルタの特徴を述べる．ここで，フィルタサイズ 3×3 のブラー処理とは，各画素とその隣接画素の画素値の平均を中心画素の画素値として新たな領域に格納する処理である．ブラー処理を分割して計算する例を**図 4** に示す．まず縦方向の平均値を計算し，それを一度バッファに格納する．その後，バッファから横方向の平均値を取り，出力

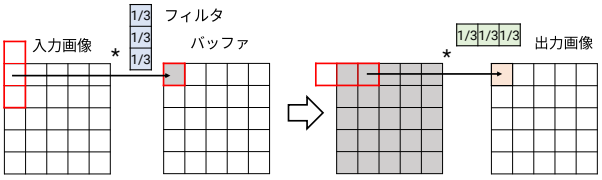

図 4 ブラー処理を分割して計算する場合

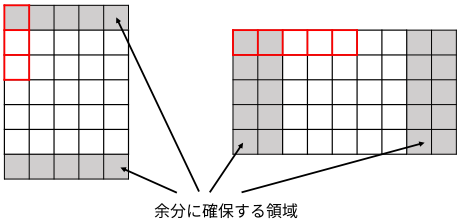

図 5 余分に確保が必要な領域

領域に書き込む．このようにセパラブルフィルタでは計算結果を保存するためのバッファが必要である．またフィルタ計算の特徴として，境界領域を処理する際にフィルタの一部が入力画像の範囲外を参照する場合がある．例えば，**図 5** のように縦方向の大きさが 3 である一次元フィルタの畳み込み演算では，縦方向の領域を入力画像の高さより上下 1 画素分ずつ大きく確保する必要がある．また横方向の大きさが 5 である一次元フィルタの畳み込み演算では，横方向の領域を入力画像の幅より左右 2 画素分ずつ大きく確保する必要がある．このようにフィルタ計算では，バッファを入力画像の幅および高さより大きく確保する必要があり，そのバッファサイズは入力画像とフィルタ計算のフィルタの大きさによって決まる．本フレームワークでは，2.2 節で述べたように，画像の幅および高さを意識せずに画像処理を記述できるプログラミングパラダイムを提供しているため，確保するバッファサイズも，専用コンパイラが判断するように実装した．通常，バッファの宣言時には，確保が必要なサイズが未知であるため，専用コンパイラはプログラムを 2 回解析する．1 回目の解析時に入力画像の幅と高さ，およびフィルタサイズを確認し，それらに基づき確保するバッファサイズを決定する．2 回目の解析時にバッファが宣言された時点でコード生成を行う．このように 2 回コードを解析することで，専用コンパイラはバッファサイズを決定することができる．これにより，バッファの宣言を記述するだけでよいため，プログラマの負担を軽減することができる．

3.2 自動最適化機能の実装

本研究では，本フレームワークの専用コンパイラに自動最適化機能を実装した．なお現状，この機能は x86 アーキテクチャにのみ対応している．専用コンパイラの概要を**図 6** に示す．まず Translator が，入力として専用言語で記述されたプログラムを受け取り，C++プログラムに

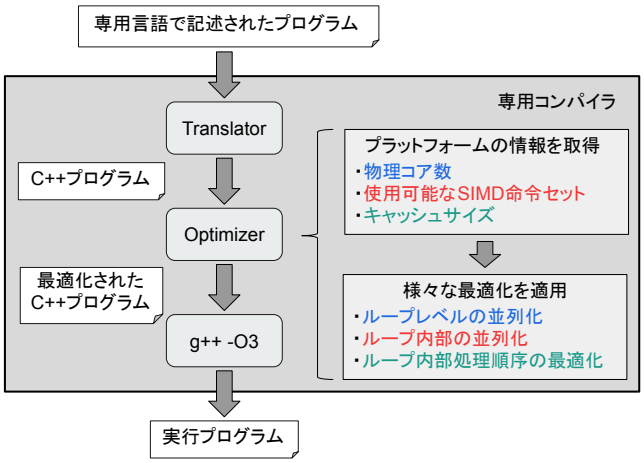

図 6　専用コンパイラの概要

変換する．この段階で，“処理単位@処理範囲” という記述はループ文に変換される．次に Optimizer が，プラットフォームの物理コア数，使用可能な SIMD 命令セット，およびキャッシュサイズを取得した後，生成した C++プログラムに対して，専用言語で記述されたプログラムの特徴に応じた最適化を施す．そして GCC の C++コンパイラを用いて実行プログラムを生成する．なお，プログラマは専用コンパイラに対し，スレッド数や SIMD 命令セットを指定することで，それらを用いて最適化したプログラムを生成することも可能である．本節では，専用コンパイラが適用する最適化の詳細について述べる．

3.2.1　ループレベルの並列化（マルチスレッド化）

一般に画像処理プログラムは高い並列性を有し，それらの多くは種々の並列処理パターン [11] を含む．例えば**マップ**や**ステンシル**と呼ばれるパターンがある．マップとは，各要素間に依存関係がなく，全ての要素に対して同じ処理を適用するようなパターンのことである．またステンシルとは，各要素とその隣接要素の値を用いて，各要素の値を更新するようなパターンのことである．Optimizer は，これらの並列処理パターンを含むプログラムに対してループレベルの並列化を施す．ループレベルの並列化では，ループ内の処理範囲を複数のブロックに分割し，それらのブロックの処理を各スレッドに割り当て，並列処理することでプログラムの高速化を図る．さらに Optimizer は，Halide では提供していない，各スレッドを別々のコアにバインドする機能も提供する．この機能によって，各コア毎に保有するキャッシュを有効活用し，プログラムの高速化を図る．

ここで，マップやステンシルを含むプログラムに対してループレベルの並列化を適用した際の効果を**図 7** および**図 8** に示す．マップおよびステンシルのワークロードとして，それぞれグレースケール変換およびブラーフィルタを用いた．どちらの図においても，画像サイズが 2048×1200 画素より大きい場合は，マルチスレッド化を適用することで，実行時間が短縮され，性能が向上する．また画像サイズが 2048×1200 画素より大きい場合において，マルチスレッド化したプログラムのコアバインド適用前後の実行時間を比較すると，スレッドをコアにバインドした方が実行時間が短く，性能が高い．これらの結果からマップおよびステンシルを含むプログラムに対しては，マルチスレッド化が有効であると考えられる．またマルチスレッド化する際に，各スレッドを別々のコアにバインドすることも有効である考えられる．なお，マルチスレッド化を適用する場合は，スレッドの生成・スケジュールに起因するオーバヘッドが発生するため，シングルスレッドで処理した方が高速になる場合があり，例えば図 7 の，画像サイズ 1024×600 の結果にその傾向が見て取れる．しかし，そのオーバヘッドを加味しても，シングルスレッドで処理した場合の実行時間と物理コアの半数のスレッドで並列処理した場合の実行時間はあまり変わらないため，マルチスレッド化に起因するオーバヘッドはわずかである．したがって Optimizer は，マップやステンシルを含むプログラムに対して，ループレベルの並列化を適用する場合は，物理コアの半数のスレッドを別々のコアにバインドして並列処理することでプログラムの高速化を図る．

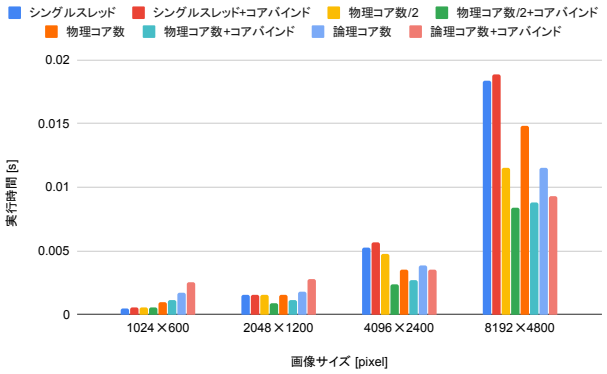

図 7　ループレベル並列化による性能向上:マップ

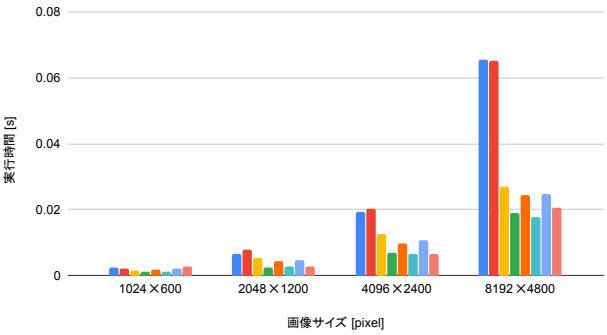

図 8　ループレベル並列化による性能向上:ステンシル

3.2.2　ループ内部の並列化（ベクトル化）

ループレベルの並列化に加え，Optimizer はマップやステンシルを含むプログラムに対して，ループ内部の処理を並列化する．ループ内部の並列化では，SIMD 命令を用い

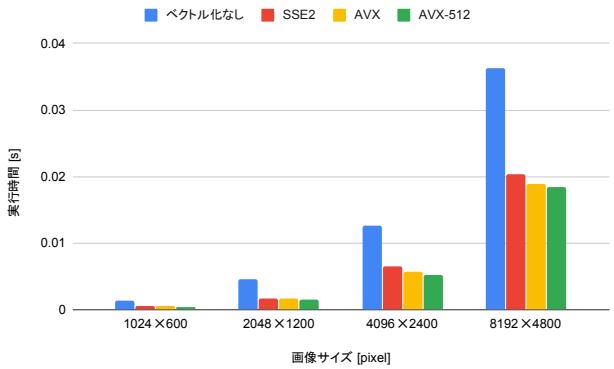

図 9 ループ内部並列化による性能向上:マップ

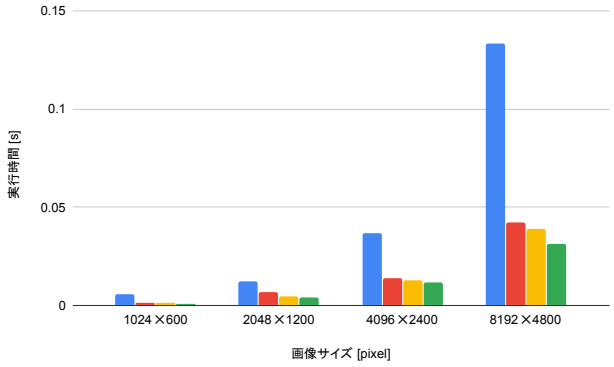

図 10 ループ内部並列化による性能向上:ステンシル

て，一度に複数の画素を処理することで，プログラムの高速化を図る．

ここで，マップやステンシルを含むプログラムに対してループ内部の並列化を適用した効果を**図 9** および**図 10** に示す．ワークロードとしてグレースケール変換およびブラーフィルタを用いた．またベクトル化には SSE2，AVX，および AVX-512 の 3 つ命令セットを用いた場合の性能を比較した．なお，SSE2，AVX，および AVX-512 の SIMD レジスタ長はそれぞれ 128，256，および 512 ビットである．どちらの図においても，ベクトル化の適用によって，プログラムの実行時間が短縮され，性能が向上している．またベクトル化に用いた SIMD 命令セットによって，プログラムの実行時間に差があることがわかる．このように性能に差がある要因として，SSE2，AVX，および AVX-512 のレジスタ長が異なる点が挙げられる．レジスタ長が大きいほど，一度に処理可能な画素数が増えるため，SSE2，AVX，AVX-512 の順にベクトル化を適用した際の性能が向上する．したがって Optimizer は，ベクトル化の適用時にプラットフォームで使用可能な SIMD 命令セットの中で最もレジスタ長の大きい SIMD 命令セットを選択し，その命令セットを用いてループ内部の処理を並列化することで，ベクトル化適用時の効果を最大限に引き出す．

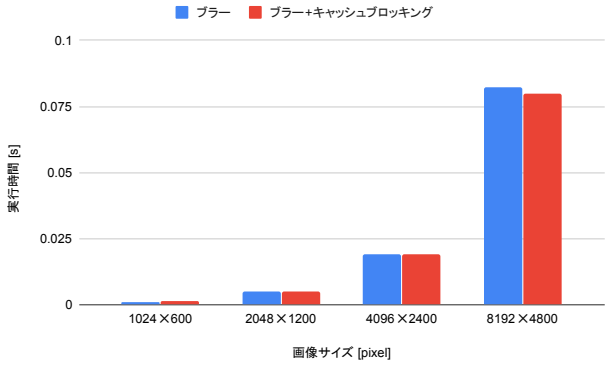

図 11 ループ内部処理順序最適化による性能向上

3.2.3 ループ内部処理順序の最適化

プログラムを高速化するためには，並列化可能な処理に対してマルチスレッド化やベクトル化を適用するだけなく，キャッシュミスを抑制することも重要である．そこで Optimizer は，ループレベルおよびループ内部の並列化に加え，ステンシルを含むプログラムに対して，ループ内部の処理順序を最適化する**キャッシュブロッキング** [12] を適用することで，プログラムの高速化を図る．キャッシュブロッキングとは，キャッシュサイズを考慮してループ内部の処理順序を変更することで，キャッシュヒット率を向上させる高速化手法である．画像処理におけるステンシルは各画素を複数回参照するため，キャッシュを有効活用できなければ，キャッシュミスが頻発する．そこで Optimizer は，ステンシルを含むプログラムに対してキャッシュブロッキングを適用することで，キャッシュミスを抑制し，プログラムの高速化を図る．

ここで，ステンシルを含むプログラムに対してループ内部の処理順序を最適化した際の効果を**図 11** に示す．ワークロードとしてはブラーフィルタを用いた．図 11 に示す通り，画像サイズが大きい場合は，キャッシュブロッキングの適用によって，プログラムの実行時間が減少し，性能が向上する．一方で，画像サイズが小さい場合は実行時間が増加し，性能が低下する．これはキャッシュブロッキングを適用する場合，ループが多段化し，実行される条件分岐命令が増加するためである．このようにキャッシュブロッキングの適用によってプログラムの性能が向上する場合と低下する場合がある．そして性能を向上させるためには，より多くのキャッシュミスを抑制できるような場合に対して，キャッシュブロッキングを適用する必要がある．したがって Optimizer は，ステンシルを含むプログラムに対して，各スレッドに割り当てられた処理範囲が L2 キャッシュサイズより大きい場合にキャッシュブロッキングを適用することで，プログラムの高速化を図る．

表 2　評価環境

	評価環境 1	評価環境 2
OS	Ubuntu 16.04	
CPU	Intel Core i9-7900X	Intel Core i7-8700K
Clock	3.30 GHz	3.70 GHz
物理/論理コア	10/20	6/12
Cache L1d/L2/L3	32/1024/14080 KB	32/256/12288 KB
Memory	64 GB	16 GB
SIMD	AVX-512	AVX2
Compiler	gcc-5.4.0	
Option	-O3	

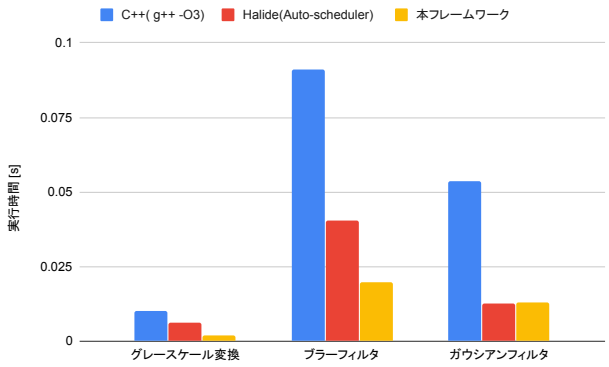

図 12　評価環境 1 におけるワークロードの比較

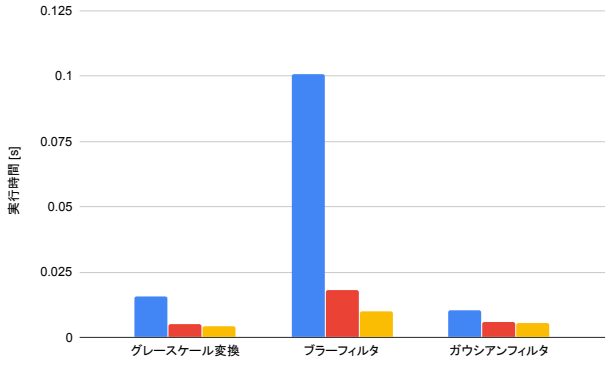

図 13　評価環境 2 におけるワークロードの比較

4. 評価

本フレームワークの有用性を検証するために，専用コンパイラが最適化したプログラムの性能を評価した．評価に用いた環境を**表 2** に示す．ワークロードとしてグレースケール変換，ブラーフィルタ，およびガウシアンフィルタの 3 つの画像処理プログラムを用いた．これらのプログラムを各 30 回実行し，それらの実行時間の中央値を C++コンパイラの O3 オプションによって最適化したプログラムおよび Halide の Auto-scheduler によって最適化したプログラムの実行時間の中央値と比較した．なお，画像サイズは 4096×2400 画素である画像を使用した．

評価結果を**図 12** および**図 13** に示す．この結果より，本フレームワークの Optimizer が最適化したプログラムは，Halide の Auto-scheduler が最適化したプログラムと比べて，平均約 1.7 倍，最大約 3.0 倍高速であることを確認した．このように Optimizer が Halide の Auto-scheduler より高性能なプログラムを生成できた要因のひとつとして，Halide で提供されていないスレッドをコアにバインドする機能を Optimizer が提供している点が挙げられる．この機能により，あるコアで実行中のスレッドがそのコアで実行できなくなった際に別コアへ移行すること (**CPU マイグレーション**) を防ぐことができる．これを防ぐことで，各コアが保有するキャッシュを有効活用できるため，プログラムの性能向上が期待できる．

表 3　スレッドをコアにバインドする効果:グレースケール変換

	Halide	本フレームワーク
CPU マイグレーション回数	59	20
L2 キャッシュミス回数	37,635	18,226
実行時間	0.00618	0.00204

ここで，評価環境 1 において，グレーススケール変換プログラムの実行中に発生した CPU マイグレーション回数および L2 キャッシュミス回数を**表 3** に示す．測定には perf コマンドを用いた．この表に示す通り，Optimizer が最適化したグレースケール変換プログラムは，Halide と比較して CPU マイグレーション回数および L2 キャッシュミス回数が少なく，キャッシュを有効活用できていることがわかる．その結果，本フレームワークの Optimizer が最適化したグレースケール変換プログラムでは，Halide の Auto-scheduler が最適化したプログラムと比較して，約 3.0 倍高速になったと考えられる．

5. おわりに

本稿では，我々が提案しているフレームワークの専用言語を拡張し，更に多くの処理を記述することが可能となった．また専用コンパイラに自動最適化機能を実装した．本フレームワークの有用性を評価した結果，専用コンパイラが最適化したプログラムは，Halide の Auto-scheduler が最適化したプログラムと比較して平均約 1.7 倍，最大約 3.0 倍高速であることを確認した．したがって計算機アーキテクチャや並列処理に精通していないプログラマでも，本フレームワークの専用言語を用いることで画像処理プログラムを簡潔に記述でき，そのプログラムを専用コンパイラが最適化することで高性能なプログラムの生成が可能である．

今後の課題として，Optimizer が行う最適化を更に追加することが挙げられる．また画像処理プログラムだけでなく，動画像処理プログラムに対する自動最適化機能を実装することが挙げられる．さらに本フレームワークを GPU，モバイルプロセッサ，および ASIC 等の様々なプラットフォームへ対応させることが挙げられる．

謝辞 本研究の一部は，JSPS 科研費 17H01711，17H01764，17K19971 の助成を受けたものである．また学際大規模情報基盤共同利用・共同研究拠点，および，革新的ハイパフォーマンス・コンピューティング・インフラの支援による（課題番号 jh190039-ISH）．

参考文献

[1] 古橋一輝，津邑公暁，“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，” 情処研報 (ETNET2018)，vol.2018-ARC-230，no.14，pp.1–6，March 2018.
[2] 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.
[3] J. Hegarty, et al., “Rigel: flexible multi-rate image processing hardware,” ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016, ACM, July 2016.
[4] 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.
[5] 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.
[6] 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.
[7] R. Baghdadi, J. Ray, M.B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, “Tiramisu: A polyhedral compiler for expressing fast and portable code,” Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, pp.193–205IEEE Press, 2019.
[8] 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.
[9] R.T. Mullapudi, et al., “Automatically scheduling halide image processing pipelines,” ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2016, ACM, 2016.
[10] 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.
[11] M. McCool, J. Reinders, and A. Robison, Structured parallel programming: patterns for efficient computation, Elsevier, 2012.
[12] M.D. Lam, et al., “The cache performance and optimizations of blocked algorithms,” ACM SIGARCH Computer Architecture News, pp.63-74, ACM, 1991.