Vol.2018-ARC-233 No.11
2018/12/7

高位合成用DSLコンパイラを用いたコーナー検出処理のハードウェア実装

原 凌司[1,a)]　井上 優良[1,b)]　谷本 輝夫[1,c)]　大澤 隆志[2]　丸岡 晃[2]　飯塚 拓郎[3]　井上 弘士[1,d)]

概要：ソフトウェアを実行する方式のプロセッサはプログラムの読み込みやデコードなどのために回路や電力を必要とする．そのため，画像処理など定型処理の実行効率を高める手段としてFPGA（Field-Programmable Gate Array）によるハードウェアアクセラレーションが注目されている．しかし，ソフトウェアプログラマにとって，HDL（ハードウェア記述言語）を用いたアプリケーションの記述は容易ではない．そこで，HDL記述を必要とせずに画像処理向けDSLであるHalideプログラムから論理合成可能なプログラムを生成するDSLコンパイラとしてGENESISが開発された．しかしながら，具体的なアプリケーションを対象としてGENESISにより生成された回路の性能は明らかではない．本研究では、OpenCVライブラリの一部をHalideで記述し，GENESISを用いて合成したFPGA用回路の性能を評価する．

1. はじめに

画像処理は様々な分野で応用されており，しばしば膨大なデータを処理する．そのため，処理速度を高める必要がある．画像処理が大量のデータに対して施される例として，画像認識が挙げられる．ここでの画像認識とは，コンピュータが画像および動画に写っているオブジェクトが何であるか，それらの持つ特徴に基づき認識する技術を指す．この技術はデジタルカメラや自動車の運転支援システムや監視カメラなどに利用されており，その認識率の向上は重要課題となっている．

現在も認識率を向上させるために，様々な方面からアプローチされており，その一つに機械学習による認識率向上がある．機械学習は大量の画像を入力とし，それらに対してアルゴリズムを適用した結果を用いてモデルを構築する．これにより，コンピュータは認識するべき画像の特徴を理解し，判断可能となる．学習の効率を上げるため，学習用画像のノイズを取り除くなどの前処理を施すことが多い．また，機械学習アルゴリズム内でも，画像圧縮手法のひとつである畳み込みなどの画像処理が施される．このように，機械学習は大量の画像に対して前処理や畳み込みなどの画像処理を行う必要があるため，膨大な量の計算を必要とする．したがって，より高速な学習のためには画像処理の高速化が必要不可欠である．

いわゆるCPUなど，ソフトウェアを実行する方式のプロセッサはプログラムの読み込みやデコードなどのために回路や電力を必要とする．そこで，画像処理など特定の処理の実行効率を高める手段として，FPGA (Field-Programmable Gate Array) によるハードウェアクセラレーションが注目されている．対象とする処理に特化した専用回路を利用できるため，ソフトウェア実行方式のプロセッサに比べ高速化が期待できる．しかしながら、FPGAを活用するためにはHDL (Hardware Description Language) によって対象となる機能を記述する必要がある．これは設計コストが非常に高く，特にソフトウェアプログラマにとっては依然として容易ではない．

そこで，画像処理向けDSL（Domain Specific Language）であるHalide [7] プログラムから，論理合成可能なプログラムを生成するDSLコンパイラGENESISが株式会社フィックスターズにより開発されている．DSLは対象とする処理を効率よく表現するために設計された言語である．GENESISを用いることで、HDL記述なしにハードウェアアクセラレーションを画像処理に適用できる．その結果，プロセッサでのソフトウェア処理に比べ，処理の高速化が期待できる．しかしながら，具体的なアプリケーションを対象としてGENESISにより生成された回路の性能は明らかではない．本研究では、OpenCV [1] の機能の一つであるFASTコーナー検出器をHalideで記述し，GENESISを用いて合成したFPGA用回路の性能を評価した．その結果，

[1] 九州大学
[2] 株式会社フィックスターズ
[3] Fixstars Solutions Inc.
[a)] ryoji.hara@cpc.ait.kyushu-u.ac.jp
[b)] yusuke.inoue@cpc.ait.kyushu-u.ac.jp
[c)] tteruo@kyudai.jp
[d)] inoue@ait.kyushu-u.ac.jp

表 1　Halide のコンパイラターゲット

CPU アーキテクチャ	X86, ARM, MIPS, Hexagon, PowerPC
GPU API	CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal, Microsoft DirectX 12

GENESIS による FPGA 上での FAST アルゴリズム実行時間は OpenCV が備える FAST アルゴリズムの実行時間より 87.4%，Halide で記述した FAST アルゴリズムの実行時間より 97.6%削減された．

本稿の構成は以下の通りである．まず第 2 節で画像処理向けの DSL である Halide について述べる．第 3 節では，Halide プログラムを入力とし，高位合成可能な C++プログラムを出力する高位合成用 DSL コンパイラである GENESIS について述べる．第 4 節で画像から特徴点の一種であるコーナーを検出する FAST アルゴリズムについて説明する．第 5 節では，実際に FAST アルゴリズムをハードウェアに実装した際の手法について説明する．第 6 節で GENESIS を用いた際の実行時間と合成されたハードウェアの観点で評価する．第 7 節で関連研究について説明し，第 8 節で本研究をまとめる．

2. 画像処理向け DSL：Halide

Halide は画像処理に特化した純粋関数型の DSL である．純粋関数型言語では関数の返り値以外でプロセスの状態が変更されない．表 1 に示すように Halide は様々なターゲットに対してコンパイルできる．一般的に，DSL は特定の処理向けの制御構造のみを持っており，汎用的な制御文を記述することは困難である．しかしながら，Halide は汎用的な制御文を表現することが可能である C++をラップする．そのため，汎用的な制御文の記述は C++同様に記述でき，柔軟な記述が可能である．

Halide はアルゴリズム部分とスケジューリング部分を分けて記述する．アルゴリズム部分には，畳み込み演算や平滑化などの計算すべき内容を記述する．スケジューリング部分では，タイリング，ベクトル化，並列化，などのハードウェアを意識した処理の最適化を指定する．このように，Halide を用いることにより，高い抽象度で記述可能なアルゴリズムと，ハードウェアを意識した最適化それぞれを分けて記述できる．

3. 高位合成用 DSL コンパイラ：GENESIS

3.1 概要

GENESIS は株式会社フィックスターズによって開発中の FPGA 向け DSL コンパイラである．画像処理向け DSL である Halide で記述されたプログラムを入力とし，Xilinx 社の Vivado HLS [10] 向けの高位合成可能な C++プログラムを出力する．Halide では，アルゴリズムが複数のデータ集合の各要素に対する副作用のない関数として記述される．この記述方式は高位合成処理系による関数から演算器への変換と親和性が高い．高位合成により関数をハードウェア・モジュール (Verilog における module) とする場合，入力ポートと出力ポートに引数と返り値をそれぞれマッピングした演算器に変換できる．GENESIS は入力された Halide プログラムのデータフローおよび制御フローを解析し，その結果に基づき後段の Vivado HLS が効率の良いハードウェアを生成できるような C++プログラムに変換する．

3.2 GENESIS による高位合成向け C++プログラム生成

一般に，ハードウェアはデータを供給・送出する I/O の性能と，供給されたデータを処理する演算器の性能が釣り合う際に効率よく動作する．GENESIS は入力された Halide プログラムを，Vivado HLS が高位合成により演算器または I/O を生成するための C++プログラムに変換する．以下では，GENESIS のプログラム変換手法について説明する．

まず，Vivado HLS により演算器が生成されるような C++記述への変換手法について述べる．GENESIS は Halide プログラムを解釈し，パイプライン化及び並列化された演算器を生成することができる．入出力の演算スループット及び I/O スループットのうち、最もボトルネックになる部分が 1 サイクルごとに演算またはデータ供給出来るようパイプライン化される．

例として，下記の `x` 方向 2 倍のダウンサンプリングを考える．

```
1  f(x, y) = in(2*x, y);
```

これは，入力画像の `x` 方向に関し，奇数番目のデータを間引く処理である．この場合，スケジューリング関数を指定しなければボトルネックは入力である `in` のデータ読み出しになり，1 サイクルにつき `x` を 1 要素読み込む．したがって，`f` は 2 サイクルにつき 1 要素演算結果を出力する．しかしながら，`in` に対して `hls_burst(2)` スケジューリングを適用してバス幅を 2 倍にすることによりボトルネックが解消され、1 サイクルにつき 1 要素出力することができる．また，演算器の並列化は Halide に用意されている `unroll` スケジューリングを用いる事で制御される．

次に，Vivado HLS によりデータ供給・送出用の I/O を生成するための C++記述の生成手法について述べる．データを供給するための I/O の方式としてメモリ I/O とストリーム I/O の 2 種類が考えられる．

メモリ I/O では，FPGA 内の SRAM あるいは FPGA 外の DRAM 上の記憶領域にデータを配置し，アドレッシング可能なメモリバスを介してアクセスする．この方式は任意のアドレスへアクセス可能なため，複雑なアクセスパターンを持つ処理を実行可能である．しかしながら，メモリ I/O を含む処理はパイプライン化できず，アクセス対象の記憶

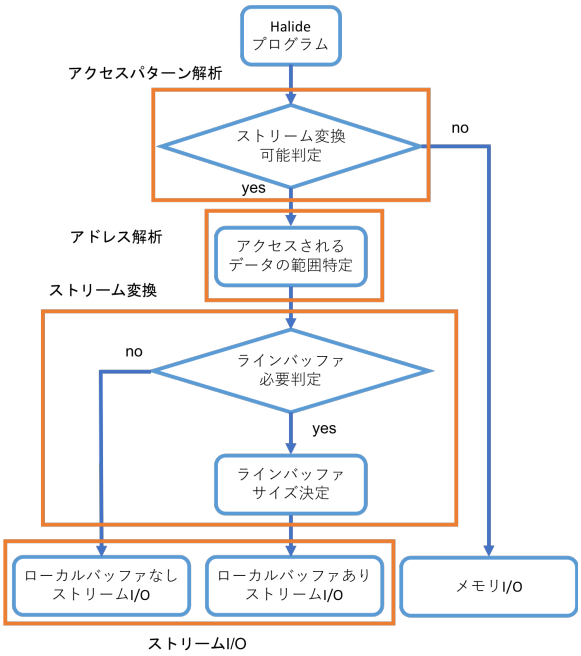

図 1 GENESIS の I/O 生成アルゴリズム．アドレス解析に基づきメモリ I/O とストリーム I/O のいずれかを生成する．

領域への全ての書き込みが完了するまで読み込みを開始できない．この理由はメモリ I/O より後段の処理が読み出すデータが，前段の処理の結果が書き込まれたものであることを保証できないためである．

一方，ストリーム I/O は連続アドレスへのデータ読み書きなど連続したデータ列を入出力するための方式である．アクセスパターンに制約を持つが，データの処理順序が明確であるため複数の処理をパイプライン化可能である．パイプライン化された処理のデータアクセスパターンが完全に同じかつアドレスが一定のストライドを伴って増加するならばストリーム I/O の適用は容易だが，一般にはそうとは限らない．

GENESIS はアドレスが一定のストライドを伴って増加するアクセスパターンをストリーム I/O として構成可能と判定する．そこで，GENESIS では柔軟なストリーム I/O を実現するために，アクセスパターン解析とアドレス解析，ストリーム変換の 3 つの手法を用いる．図 1 に，I/O 生成アルゴリズムを示す．まず，アクセスパターン解析によって，変換対象アルゴリズム内の各バッファがストリーム変換可能かどうかを判定する．アクセスパターンが逐次アクセスに準じるパターンであるバッファはストリーム変換可能と判定され，それ以外はメモリ I/O に変換される．

次に，ストリーム変換可能と判定されたバッファアクセスに対してアドレス解析を行い，ストリーム変換された場合における参照範囲を特定する．アドレス変換の結果に基づき，レジスタを用いたラインバッファを構築し，そこに当該データを格納する．このラインバッファは静的なアドレッシングが可能となるように最適化される．アクセスが必要な際にはクロスバースイッチが操作され，バッファから演算器へデータが転送される．

最後に，ストリーム I/O に対する入出力処理とラインバッファとのデータ供給に関する処理を生成し，間断なく演算器にデータ供給可能なデータストリームを構築する．GENESIS は可能な限りストリーム I/O に変換するようにコード生成を行うが，Halide のスケジューリング関数を通じてユーザが I/O の種別を指定することも可能である．

4. FAST アルゴリズムを用いたコーナー検出

4.1 FAST アルゴリズム

FAST（Feature from Accelerated Segments Test）は Rosten と Drummond によって提案された，特徴点の一種であるコーナーを検出する手法である [8], [9]．コーナーは輝度の変化が大きい角もしくはその周辺の画素に現れる．FAST によるコーナー検出アルゴリズムは以下の通りである．

1. ある注目画素を基準として定め，それを中心とする半径数ピクセル分離れた円周上の各画素に着目する．

2. 注目画素に対し，円周上の各画素の輝度値がより高い（明るい），低い（暗い），または，同程度のいずれに分類されるかを式 (1) により判定する．

$$\begin{cases} brighter & (I_x > I_p + t) \\ darker & (I_x < I_p - t) \\ samelevel & \text{上記以外} \end{cases} \tag{1}$$

ここで，I_x は円周上から選択された画素の輝度，I_p は注目画素の輝度，t は閾値を表す．

3. 上記 1 で定めた円周上において，brighter または darker と判定された画素が指定数以上検出された場合にコーナーであると判定する．

このアルゴリズムの問題点はコーナーであると判定した画素に隣接した画素もコーナーである場合が多く，冗長にコーナーを検出することである．しかし，この問題点は検出した各コーナーに，以下の式 (2) で算出される値をスコアとして導入する事で解消される．

$$score = max\left(\sum_{x \in brighter} |I_x - I_p| - t, \sum_{x \in darker} |I_x - I_p| - t \right) \tag{2}$$

隣接し合っているコーナーのうち，スコアが最も大きなコーナー以外を棄却することで，上記問題は解消される．

4.2 実行例

白黒の正方形が並べられている画像を対象とし，OpenCV が備える FAST コーナー検出器によりコーナー検出を行っ

図 2　OpenCV がそなえる FAST コーナー検出器によるコーナー検出結果

		15	0	1		
	14				2	
13						3
12			p			4
11						5
	10				6	
		9	8	7		

図 3　FAST によるコーナー検出の際に参照されるピクセルの配置

た結果を図 2 に示す．OpenCV による FAST コーナー検出器はコーナーであると判定した画素を丸で囲み表示する．図 2 を見ると，正方形の角にコーナーが集中している事を確認できる．

FAST は注目画素の円周上に，注目画素より明るい画素が複数個連続もしくは暗い画素が複数個連続した場合にコーナーと判定する．その特性上，明暗が切り替わる角に当たる画素をコーナーとする．この実行では，図 3 のような半径 3 ピクセルの円周上 16 ピクセルに，注目画素より 9 個連続して明るいもしくは暗い画素が存在する場合にコーナーと判定するように設定した．並べられている四角は画素を表している．画素 p はコーナー候補であり，その候補を中心とする半径 3 ピクセルの円周上の画素 0，1,...，15 がコーナー判定の計算に用いられる．

5. FAST アルゴリズムのハードウェア実装

5.1 Halide によるアルゴリズム記述

まず，特徴点を検出する対象となる画像をファイルから取得する必要がある．これは Halide::Tools が備えている load_image を使用することで，Halide 記述に適した形で取得できる．

```
1  Halide::Runtime ::Buffer<uint8_t> __img =
       load_image("<path>");
```

また，式として使用するために，Halide 特有の型である，Func 型に取得画像を格納する．

```
2  Var x,y,c;
3  Func _img;
4  _img(x,y,c) = __img(x,y,c)
```

_img の x，y，c 座標に，__img の x，y，c 座標の画素値を順に格納することで，画像全体を格納する．上記の式の x,y,c は Halide 特有の Var 型で宣言する必要がある．FAST アルゴリズムでは注目画素を中心とする円周上の画素にアクセスする特性上，画像領域外の画素にアクセスする事がある．この場合に備え，Halide では BoundaryCondition を用いることで，領域外の参照値を設定できる．本実装では，画像端を画像で折り返すよう設定した．また，コーナー検出に用いる参照範囲を半径 3 ピクセルの円周とし，その円周上には 16 個の画素があるものとした．

```
5  img = BoundatyConditions::mirror_interior(_img, "
       <imagesize>");
```

本実装において，注目画素は画像端から順に選択した．Halide では式を変数として定義する．

```
6  Expr value = img(x,y,c);
7  Expr value0 = img(x,y+3,c);
```

また，円周上の画素値を取得する必要があるため，7 行目のような式を円周上のピクセルの数用意する．注目画素の画素値と円周上の画素の画素値を比較する式には Halide の関数である select を用いた．

```
8  Expr comn0 = select(value - threshold > value0,
       -1, 0);
```

select は（条件文, 式 1, 式 2）の形をとる．条件文が真であるなら式 1，偽であれば式 2 が参照される．threshold は閾値を表す定数である．上記の場合，注目画素値から閾値を引いた値より，円周上の画素が暗い場合は-1，そうでない場合は 0 が参照される．このような式を円周上の画素数用意する．また，円周上の画素の方が注目画素より明るい場合の式も必要である．明るい場合を表す式を以下に示す．

```
9  Expr comp0 = select(value + threshold < value0,
       1, 0);
```

注目画素がコーナーであると判定されるのは円周上の画素が連続して指定数個注目画素より明るいもしくは連続して指定数個注目画素より暗い時である．本実装では，9 個連続する場合コーナーとした．

```
10 Expr cornern = select(cast<int8_t>((comn0 + comn1
       + ... + comn8)) == -9, 1, (cast<int8_t>(
       comn1 + comn2 + ... + comn9)) == -9, 1,
       ...,(cast<int8_t>(comn15 + comn0 + ... +
       comn7)) == -9, 1,0);
```

comn0，...，comn15 は式であり，それぞれ注目画素より暗い場合は-1, そうでない場合は 0 の値を返す．円周上の点 0 に対応する comn0 から始め，そこから点 8 に対応する comn8 までの 9 画素分の総和が-9 であれば，9 個連続して暗い場合であるので，コーナーと判定する．また，円周上の画素が 9 個連続して明るい場合もコーナーであると判定するため，上記の式と同様な式を立てる必要がある．次

に，コーナーの冗長な検出を棄却するための式を考える．スコアの計算には，コーナーの円周上の，明るいもしくは暗い画素を参照して計算をする必要がある．

```
11 Expr scoreelem_n0 = select(comn0 == -1, abs(cast<
       int8_t>(value0 - value)) - threshold, 0);
```

この式はコーナーの円周上の 1 画素が暗い場合の計算を施す．この式を円周上の画素分用意する．また，明るい場合の計算式も同様に用意する．上記の式で，円周上の画素を用いたスコア計算に必要な値が揃うので，それらを総和し，スコアとする．

```
12 Expr score_n = scoreelem_n0 + ... + scoreelem_n15
13 Expr score_p = scoreelem_p0 + ... + scoreelem_p15
14 Expr score = select(score_n > score_p, score_n,
       score_p);
```

また，注目画素の輝度値が閾値未満もしくは，255 - threshold より大きい場合，円周上の明るい画素や暗い画素を探す際の計算などでオーバーフローが発生する．その結果，コーナーでない画素をコーナーと誤検出する．この問題を解消するために，注目画素が上記の値の場合，コーナーでないとする必要がある．以下がそのための式である．

```
15 Expr _cornern = select(value >= threshold,
       cornern, 0);
16 Expr _cornerp = select(value <= (255 - threshold)
       , cornerp, 0);
17 Expr result1 = select(_cornern == 0 && _cornerp
       == 0, 0, score);
```

_cornern には注目画素が閾値以下でない場合，円周上に暗い画素が 9 個連続するパターンでのコーナー検出の結果が反映され，コーナーであるとき 1，そうでないとき 0 となる．また，cornerp は注目画素が 255-threshold を超えていない場合，円周上に明るい画素が 9 個連続するパターンでのコーナー検出の結果が反映され，コーナーであるとき 1，そうでないとき 0 となる．注目画素が閾値より暗い場合，_cornern は 0 であり，_cornerp 次第でコーナーか判断される．また，注目画素が 255-閾値より明るい場合，_cornerp は 0 となり，_cornern 次第でコーナーか判断される．スコアを使って冗長なコーナーを棄却する式は次のようになる．

```
18 Func Fast;
19 Fast(x,y,c) = result1;
20 Func output1;
21 output1 = BoundaryConditions::mirror_interior(
       Fast, {{0, __img.width()}, {0, __img.height
       ()}});
22 Expr comscore = select(output1(x,y,c) < output1(x
       ,y+1,c) || output1(x,y,c) < output1(x+1,y,c)
        || output1(x,y,c) < output1(x,y-1,c) ||
       output1(x,y,c) < output1(x-1,y,c), 0,
       output1(x,y,c));
23 Expr _comscore = select(nonmax_suppression ==
       true, comscore, output1(x, y, c));
24 Func sup;
25 sup(x,y,c) = _comscore;
```

上下左右の隣接する画素を比較する必要があるため，領域外アクセスの可能性がある．よって，BoundaryCondition を使用する．comscore は注目しているコーナーが隣接するコーナーのうち，少なくとも 1 つよりスコアが劣る場合にスコアを 0 とする事で棄却する．_comscore は棄却する処理をするかどうかを nonmax_suppression で判断し，true の場合棄却処理の結果が，false の場合棄却処理の前の結果が反映される．

5.2 GENESIS におけるスケジューリング記述

GENESIS ではスケジューリング部分での記述により，演算器の展開数や I/O のバス幅などのアーキテクチャパラメータを指定することができる．たとえば，unroll(x, n) スケジューリングを用いる場合，x 方向の処理を n 分割し，n 倍の演算器を生成することで並列処理する．ただし演算器の生成のみでは入出力のスループットがボトルネックになり性能は向上しないため，入出力のバス幅も n 倍する必要がある．そのため，hls_burst(n) スケジューリングにより，入出力のバス幅を n 倍にする．本稿では n のことをバースト数と呼ぶ．

5.3 合成されたハードウェア

FAST アルゴリズムを実装した Halide プログラムを GENESIS により高位合成可能な C++に変換し，さらに Vivado HLS，Vivado により合成されたハードウェアの構成を図 4 に示す．中央に位置する orb_detect_0 は FAST アルゴリズムにより特徴点を検出する回路である．orb_detect_0 と繋がっているモジュールに rst_ps7_0_100M，processing_system7_0，ps7_0_axi_periph，そして axi_mem_intercon がある．rst_ps7_0_100M は orb_detect_0 へリセット信号を送る．processing_system7_0 はプロセッシングシステム周辺のソフトウェアインタフェースである．axi_mem_intercon は orb_detect_0 と processing_system7_0 が備えるメモリとのインタフェースである．DRAM から読み出されたデータは S00_AXI を経由し m_axi_p_img で渡され，DRAM への書き込みデータは m_axi_p_sup から，S01_AXI を経由して渡される．ps7_0_axi_periph は zybo-z7-20 に接続されている USB や LAN などの周辺機器とのインタフェースである．

図 5 に orb_detect_0 の構成を示す．image は DRAM から読み出した画像を確保する．fast は image に確保されている画像を受け取り，FAST アルゴリズムによりコーナーを検出する．sup は冗長に検出したコーナーを棄却する．

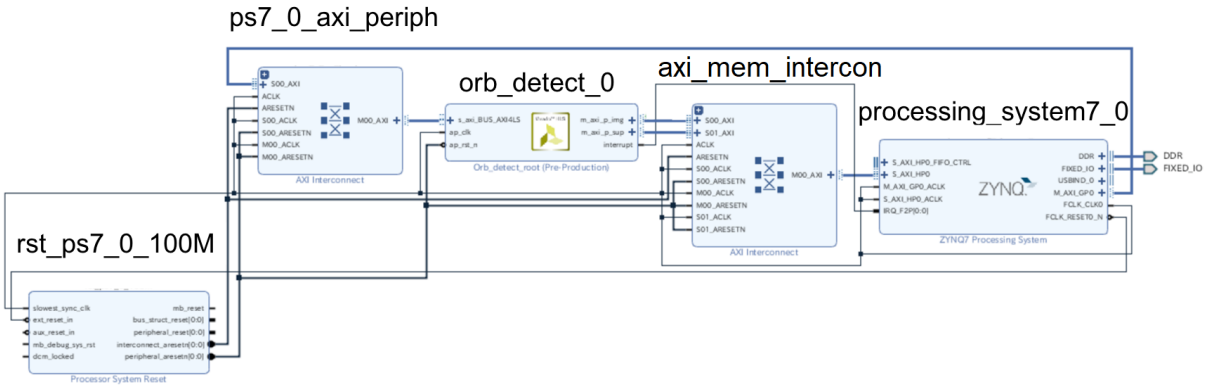

図 4　GENESIS を用いて合成されたハードウェアのブロック図

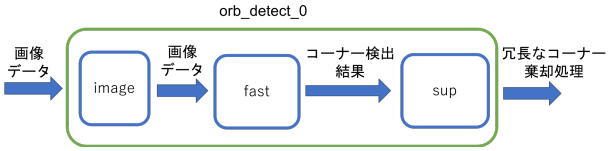

図 5　orb_detect_0 の構成

表 2　評価に用いたハードウェア環境（Zynq-7020）

CPU	667MHz dual-core Cortex-A9 processor
L1 命令キャッシュ	32 KB（各コアで独立）
L1 データキャッシュ	32 KB（各コアで独立）
L2 データキャッシュ	512 KB （コア間で共有）
ロジックスライス	13,300
6 入力 LUT	53,200
FF	106,400
ブロック RAM	630 KB
メモリ	1 GB DDR3L with 32-bit bus @ 1,066 MHz

表 3　評価に用いたソフトウェア環境

OS	Linux 4.9.0
C++ コンパイラ	GCC 4.8.3
Vivado バージョン	2018.2
Vivado HLS バージョン	2018.2

6. 評価

6.1 評価の概要

本稿では，Halide で記述された FAST アルゴリズムによるコーナー検出プログラムを，GENESIS を用いてハードウェア化し，FPGA 上で実行した際の実行時間を計測する．そして，CPU 上での OpenCV が備える FAST アルゴリズムによるコーナー検出プログラムの実行時間，CPU 上での Halide で記述した FAST アルゴリズムによるコーナー検出プログラムの実行時間と比較することで評価する．評価に用いたハードウェア環境を表 2 に，ソフトウェア環境を表 3 に示す．

表 4　ハードウェア資源使用量と使用率

	使用量	使用率
6 入力 LUT	16,282	31%
FF	15,804	20%
ブロック RAM	37.8 KB	6%

6.2 ハードウェア合成結果評価

Halide で記述された FAST アルゴリズムによるコーナー検出プログラムを，GENESIS を用いて FPGA 上に実装した場合のハードウェア資源使用量とその使用率を表 4 に示す．なお，本実験ではバースト数を 8 としている．

ハードウェア資源使用量は半分以下であることが分かる．このことから，バースト数を 16 に変更しても FPGA 上に FAST アルゴリズムを実装できることが予測される．よって，FAST アルゴリズムは FPGA 上でより高速に動作できる可能性がある．

また，配置配線後の最大遅延は 8.75 ns であった．このことから，今回生成した回路の最大動作周波数は 114.3 MHz である．ただし，以降の評価では FPGA を 100 MHz で動作させた．

6.3 実行時間評価

Halide で記述された FAST アルゴリズムによるコーナー検出プログラムを GENESIS を用いて FPGA 上で実行した際の実行時間，CPU 上での OpenCV が備える FAST アルゴリズムによるコーナー検出プログラムの実行時間，CPU 上での Halide で記述した FAST アルゴリズムによるコーナー検出プログラムの実行時間を図 6 に示す．

GENESIS を用いて，FPGA 上で FAST アルゴリズムを実行した場合の実行時間は OpenCV での FAST アルゴリズム実行での実行時間より 87.4%，Halide での FAST アルゴリズム実行での実行時間より 97.6%削減された．また，Vivado の ChipScope 機能を使って，図 4 で示した orb_detect_root が入力を読み出すメモリインタフェースの信号をプローブした．その波形を図 7 に示す．図 7 はメモリインタフェー

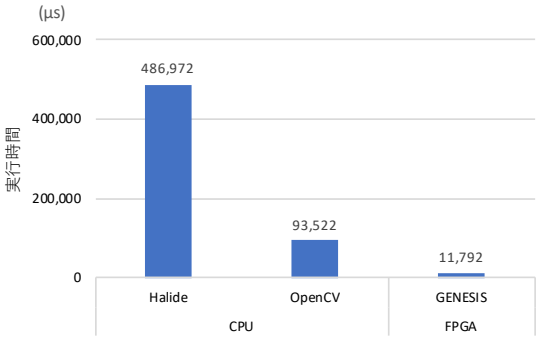

図 6 実行時間評価結果

スの周期的なメモリアクセスの一周期に相当し，134 clk 周期で 256 Byte のメモリ読み出しが発生している．今回対象とした画像は $1,920 \times 1,080$ 画素のグレイスケール（$1Byte/1$ 画素）であり，ハードウェアのクロック周期は 10 ns/clk であることを考慮すると，メモリ読み出しにかかる時間は次の式 (3) で算出できる．

$$\frac{1,920 \times 1,080 \times 1\,Byte}{256\,Byte} \times 134\,clk \times 10\,ns/clk = 10854\,\mu s \tag{3}$$

よって，実行時間の大半を占めていることが確認できる．また，この FPGA のメモリバス帯域は $4\,Byte/clk$ である．よって，今回 FPGA に実装されたハードウェアのメモリバス帯域使用率は式 (4) から

$$\frac{256\,Byte/134\,clk}{4\,Byte/clk} \times 100 = 47.7\% \tag{4}$$

であることがわかる．また，FPGA 上の FAST の実行時間が $11792\,\mu s$ であり，メモリ読み出しにかかる時間が $10854\,\mu s$ であることから，メモリ読み出し，FAST 処理，メモリ書き込みはパイプライン化されていると考えられる．よって，次の式 (5) でレイテンシは算出される．

$$11,792\,\mu s - 10,854\,\mu s = 932\,\mu s \tag{5}$$

このことから，FPGA 上で実装された FAST アルゴリズムはメモリ転送に挟まれないように処理に組み込まれることで，より性能を発揮できると考えられる．

7. 関連研究

7.1 Haskell の組み込み DSL を用いた高位合成

Haskell の組み込み DSL を設計言語とする高位合成が提案された [3]．Haskell は純粋関数型言語であり，抽象度が高い．そのため，ハードウェア設計との親和性が高い．この提案手法では Haskell の組み込み DSL を入力とし，LLVM [4] の中間表現である LLVM IR を生成する．LVM IR は LLVM の標準的な最適化がされる．最適化された LLVM IR は高位合成ツール LegUp [2] の入力となる．LegUp は LLVM IR を合成し，RTL 記述である Verilog を生成する．しかし，この提案手法ではループタイリングなどのスケーラブルな実装ができない．一方 GENESIS はループタイリングやベクトル化などのスケジュール記述が容易に可能である Halide を用いるため，上記は問題とならない．

7.2 ScalaHDL: Express and test hardware designs in a Scala DSL

ScalaHDL [5] は複数のプログラミングパラダイムに対応するマルチパラダイムプログラミング言語である Scala [6] 上に構築された DSL である．これを用いることで，低レベルのハードウェア抽象化を用いてアルゴリズムを記述し，シミュレーション及び Verilog を生成することができる．ScalaHDL には次の特徴がある．

- Scala 以外の環境をインストール不要．
- インタラクティブに実行してテストすることが可能．
- モジュールを定義する際に，入力／出力レジスタの型を自動的に推測．

これらの機能により，ユニットテスト及び完全なシステムテストシナリオを用いてプログラムで値をテストできるようにシミュレートすることを可能とする．また，複雑な環境構築を必要とせずにハードウェアシミュレーションできる．

一方で，ScalaHDL で使用できる変数の型は Bool, Signed, Unsigned である．Halide では Bool, Signed, Unsigned に加え，画像データを格納できる Buffer 型や，画像処理アルゴリズムを記述する Func 型，組み立てた式を保持する Expr 型を利用できる．これらにより，容易に画像処理アルゴリズムを記述可能である．このことから，画像処理を対象とする場合 Halide を用いる GENESIS が有効であると考えられる．

8. おわりに

本稿では，FPGA を対象とした画像処理のハードウェア実装を容易にする GENESIS の評価を行った．具体的には FAST 特徴点検出を対象に，実際に Halide でプログラムを作成し実施した．その結果，GENESIS を用いることで容易に，実行時間をソフトウェアでの実行と比較し大幅に削減できることを明らかにした．GENESIS を用いた FAST コーナー検出アルゴリズムの実行時間は CPU 上での OpenCV が備える FAST アルゴリズムの実行時間から 87.4%，CPU 上での Halide で記述された FAST アルゴリズムの実行時間から 97.6%削減された．

また，FPGA に実装される FAST アルゴリズムはメモリ転送に挟まれないように処理に組み込まれることで，さらに性能を発揮できる見込みがあることを示した．今後は FPGA 向けにチューニングされた FAST 実装との性能比較や，他の画像処理アルゴリズムを対象とした GENESIS の

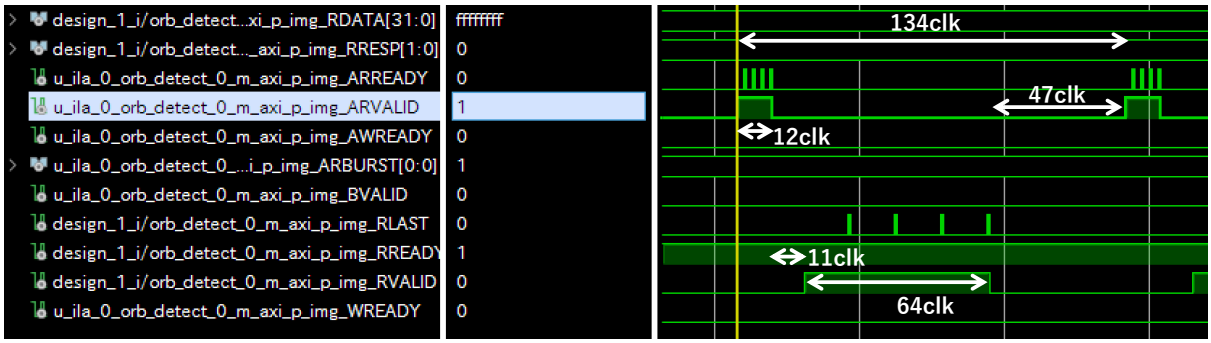

図 7 メモリインタフェースによるメモリアクセス時の信号

評価を行う．

謝辞 本研究は一部 JSPS 科研費 JP17K19984 の助成を受けたものである．

参考文献

[1] Bradski, G.: The OpenCV Library, *Dr. Dobb's Journal of Software Tools* (2000).

[2] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S. D. and Anderson, J. H.: LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems, *ACM Transaction on Embedded Computing Systems*, Vol. 13, No. 2, pp. 24:1–24:27 (online), DOI: 10.1145/2514740 (2013).

[3] Kuga, M., Fukuda, K., Amagasaki, M., Iida, M. and Sueyoshi, T.: High-level Synthesis Based on Parallel Design Patterns Using a Functional Language, *In Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART '17, New York, NY, USA, ACM, pp. 23:1–23:6 (online), DOI: 10.1145/3120895.3120918 (2017).

[4] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp. 75–86 (2004).

[5] Li, Y., Lopes, A. R., Xu, Z., Qi, Z. and Guan, H.: ScalaHDL: Express and test hardware designs in a Scala DSL, *In Proceedings of the IEEE 32nd International Conference on Computer Design*, ICCD '14, pp. 521–524 (online), DOI: 10.1109/ICCD.2014.6974732 (2014).

[6] Odersky, M., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M. and et al.: An overview of the Scala programming language, Technical report (2004).

[7] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. and Amarasinghe, S.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, pp. 519–530 (online), DOI: 10.1145/2491956.2462176 (2013).

[8] Rosten, E. and Drummond, T.: Fusing points and lines for high performance tracking., *IEEE International Conference on Computer Vision*, ICCV '05, Vol. 2, pp. 1508–1511 (online), DOI: 10.1109/ICCV.2005.104 (2005).

[9] Rosten, E. and Drummond, T.: Machine learning for high-speed corner detection, *In Proceedings of the 9th European Conference on Computer Vision*, ECCV '06, Vol. 1, pp. 430–443 (2006).

[10] Xilinx: *UG902 - Vivado Design Suite User Guide: High-Level Synthesis (ver2018.2)* (2018).