XAPP1167 (v3.0) 2015 年 6 月 24 日

Vivado HLS ビデオ ライブラリを使用して Zynq-7000 All Programmable SoC で OpenCV アプリケーションを高速化

著者 : Stephen Neuendorffer、Thomas Li、Devin Wang

概要

このアプリケーション ノートでは、OpenCV ライブラリを使用して Zynq®-7000 All Programmable SoC でコンピューター ビジョン アプリケーションを開発する方法について説明します。OpenCV は、アルゴリズムのプロトタイピングからインシステムでの実行まで、設計プロセスのさまざまな段階で使用できます。OpenCV コードは、Vivado® 高位合成 (HLS) に付属のビデオ ライブラリを使用して、合成可能な C++ コードに移行することも可能です。合成されたブロックを Zynq SoC デザインに統合することで、高い解像度とフレーム レートのコンピューター ビジョン アルゴリズムがインプリメントできます。

はじめに

コンピューター ビジョンの分野には、製造不良品を検出する産業用監視システムから自動車の運転支援システムまで、数多くの興味深いアプリケーションが広く存在します。コンピューター ビジョン システムの多くは、デスクトップ プロセッサおよび GPU をターゲットとする多数の一般的なコンピューター ビジョン機能の最適化されたインプリメンテーションを含むライブラリである OpenCV を使用してインプリメントまたはプロトタイピングされています。OpenCV ライブラリ内の多数の関数は、さまざまなコンピューター ビジョン アプリケーションをほぼリアルタイムで実行できるように高度に最適化されていますが、最適化されたインプリメンテーションが組み込まれている方が望ましい場合が少なくありません。

このアプリケーション ノートでは、OpenCV プログラムを Zynq デバイスへリターゲットするデザイン フローを示します。このデザイン フローでは、Vivado Design Suite の HLS テクノロジと、最適化済みの合成可能ビデオ ライブラリを使用します。ライブラリは直接使用することも、それをアプリケーション独自のコードと組み合わせて、特定のアプリケーション向けにカスタマイズされたアクセラレータを作成することもできます。このデザイン フローでは高性能で低消費電力の多数のコンピューター ビジョン アルゴリズムを迅速にインプリメントできます。また、高データ レートのピクセル処理タスクはプログラマブル ロジックで実行し、低データ レートのフレームベース処理タスクは従来どおり ARM® コアで実行するように設計することも可能です。

次の図に示すように、OpenCV はビデオ処理システムのさまざまな設計段階で使用できます。左側では、ファイル アクセス関数を用いるイメージの入力/出力および処理の両方に OpenCV 関数呼び出しを使用し、アルゴリズムを設計およびインプリメントできます。次に、このアルゴリズムを (Zynq ベース TRD などの) エンベデッド システムにインプリメントし、プラットフォーム独自の関数呼び出しを使用する入力/出力イメージにアクセスします。この場合、ビデオ処理は、これまでどおり (Zynq プロセッシング システム内の Cortex™-A9 プロセッサ コアなどの) プロセッサで実行される OpenCV 関数呼び出しを使用してインプリメントされます。あるいは、OpenCV 関数呼び出しを、対応する Vivado HLS ビデオ ライブラリ内の合成可能な関数で置き換えることもできます。この場合も、OpenCV 関数呼び出しは、入力/出力イメージへのアクセスと、ビデオ処理アルゴリズムの完全なリファレンス インプリメンテーションの提供に使用可能です。合成後、処理ブロックは Zynq プログラマブル ロジックに統合できます。統合されたブロックは、プログラマブル ロジックにインプリメントされるデザインに応じて、プロセッサによって作成されるビデオ ストリーム (ファイルから読み出されるデータなど) や外部入力からの生のリアルタイム ビデオ ストリームを処理します。

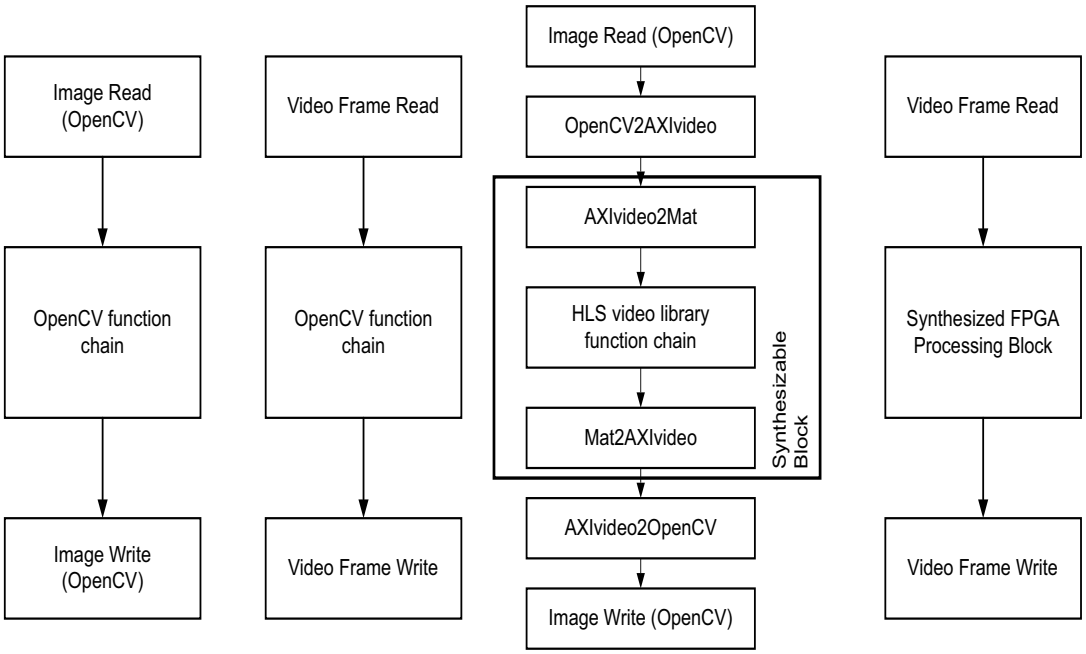

図 1 : デザイン フロー

このアプリケーション ノートの一般的なデザイン フローは、次のとおりです。

1. デスクトップで OpenCV アプリケーションを開発し、実行する。
2. OpenCV アプリケーションを変更せずに Zynq SoC 内で再コンパイルし、実行する。
3. I/O 関数を使用して OpenCV アプリケーションをリファクタリングし、アクセラレータ関数をカプセル化する。
4. OpenCV 関数呼び出しを、アクセラレータ関数内の合成可能ビデオ ライブラリ関数呼び出しで置き換える。
5. Vivado HLS を使用して、アクセラレータ関数からアクセラレータおよびそれに対応する API を生成する。
6. アクセラレータ関数の呼び出しを、アクセラレータ API の呼び出しで置き換える。
7. 高速化されたアプリケーションを再コンパイルし、実行する。

リファレンス デザイン

リファレンス デザインは、次のリンクからダウンロードできます。

https://secure.xilinx.com/webreg/clickthrough.do?cid=323570

表 1 に、リファレンス デザインの詳細を示します。

表 1 : リファレンス デザインの詳細

パラメーター	説明
全般	
開発者	Thomas Li
ターゲット デバイス (ステッピング レベル、ES、プロダクション、スピード グレード)	XC7020-1
ソース コードの提供	あり
ソース コードの形式	C

表 1 : リファレンス デザインの詳細

パラメーター	説明
既存のザイリンクス アプリケーション ノート/リファレンス デザイン、CORE Generator ツール、サードパーティからデザインへのコード/IP の使用	あり (Zynq ベース TRD アプリケーション ノート/リファレンス デザイン、CORE Generator ツール、またはサードパーティのシミュレーションに基づく)
シミュレーション	
論理シミュレーションの実施	あり、C を使用
タイミング シミュレーションの実施	なし
論理シミュレーションおよびタイミング シミュレーションでのテストベンチの利用	あり
テストベンチの形式	C
使用したシミュレータ/バージョン	g++
SPICE/IBIS シミュレーションの実施	なし
インプリメンテーション	
使用した合成ツール/バージョン	Vivado 2014.4
使用したインプリメンテーション ツール/バージョン	Vivado 2014.4
スタティック タイミング解析の実施	あり
ハードウェア検証	
ハードウェア検証の実施	あり
使用したハードウェア プラットフォーム	ZC702

Vivado HLS のビデオ処理ライブラリ

Vivado HLS には、各種のビデオ処理機能の作成を容易にするビデオ ライブラリが多数含まれています。これらのライブラリは合成可能な C++ コードとしてインプリメントされ、OpenCV にインプリメントされているビデオ処理関数およびデータ構造にほぼ対応しています。Vivado HLS におけるビデオの概念および抽象化の多くは、OpenCV のそれらと非常に類似しています。特に、OpenCV imgproc モジュールの関数の多くに、対応する Vivado HLS ライブラリ関数があります。

たとえば、OpenCV の中心的な要素である cv::Mat クラスは、通常はビデオ処理システムでイメージを表す際に使用されます。cv::Mat オブジェクトは、通常は次の例に示すように宣言されます。

```
cv::Mat image(1080, 1920, CV_8UC3);
```

これは変数 `image` を宣言し、この変数を初期化して 1080 行、1920 列のイメージを表します。ここで、各ピクセルは 3 個の符号なし 8 ビット数で表現されます。合成可能ライブラリには、これに対応する hls::Mat<> テンプレート クラスが含まれ、同様の概念を合成可能な方法で次のように表現します。

```
hls::Mat<2047, 2047, HLS_8UC3> image(1080, 1920);
```

得られるオブジェクトはほぼ同じですが、コンストラクター引数に加えてテンプレート パラメーターを使用してイメージの最大サイズとフォーマットが記述される点が異なります。これにより、Vivado HLS は、このイメージの処理に使用するメモリのサイズを決定し、得られた回路を特定のピクセル表現方法に合せて最適化できます。また、hls::Mat<> テンプレート クラスでは、処理されるイメージの実際のサイズが最大サイズと同じである場合、コンストラクター引数を完全に省略できます。

```
hls::Mat<1080, 1920, HLS_8UC3> image();
```

同様に、OpenCV ライブラリでは、`cvScale` 関数でイメージの各ピクセル値にリニア スケーリングを適用可能です。この関数は次のように呼び出されます。

```
cv::Mat src(1080, 1920, CV_8UC3);
cv::Mat dst(1080, 1920, CV_8UC3);
cvScale(src, dst, 2.0, 0.0);
```

この関数呼び出しは、入力イメージ src のピクセルをオフセットなしで 2 倍にスケーリングし、出力イメージ dst を生成します。合成可能ライブラリでこれに対応する動作は、hls::Scale テンプレート関数の呼び出しによって次のようにインプリメントされます。

```
hls::Mat<1080, 1920, HLS_8UC3> src;
hls::Mat<1080, 1920, HLS_8UC3> dst;
hls::Scale(src, dst, 2.0, 0.0);
```

注記 : hls::Scale はテンプレート関数ですが、テンプレート引数は src および dst の宣言内のテンプレート引数から推測されるため、指定する必要はありません。ただし、hls::Scale テンプレート関数では、入力イメージと出力イメージのテンプレート引数が同一でなければなりません。サポートされている関数の一覧は、『Vivado Design Suite ユーザー ガイド : 高位合成』(UG902) [参照 4] に記載されています。

ビデオ処理アーキテクチャ

Zynq SoC のビデオ処理デザインは、通常は次に説明する 2 つの一般的なアーキテクチャのいずれかに従います。「ダイレクト ストリーミング」と呼ばれる第 1 のアーキテクチャの場合、ピクセル データは、プログラマブル ロジックの入力ピンに到着した後、ビデオ処理コンポーネントに直接送られ、そこからビデオ出力へ直接伝送されます。このアーキテクチャは、一般的に最も簡単かつ効率的なビデオ処理方法ですが、ビデオ処理コンポーネントが厳密にリアルタイムでフレームを処理できることが要件です。

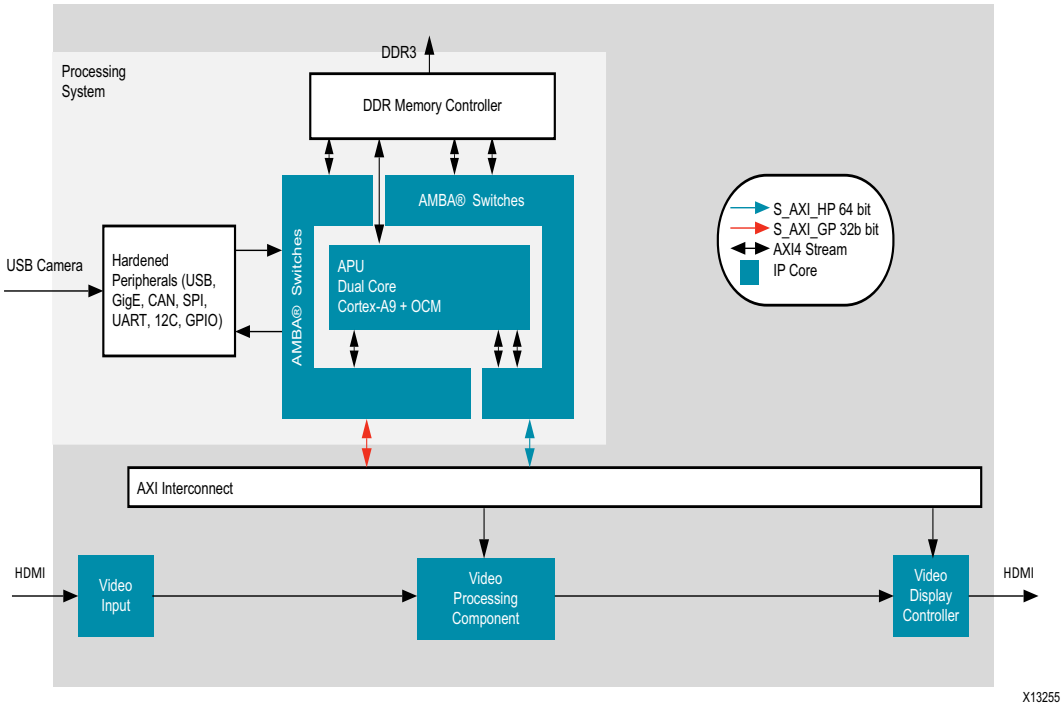

図 2 : ビデオ処理のダイレクト ストリーミング アーキテクチャ

「フレーム バッファー ストリーミング」と呼ばれる第 2 のアーキテクチャの場合、ピクセル データはまず外部メモリに格納され、処理後に再び外部メモリに格納されます。処理されたビデオの出力にはビデオ ディスプレイ コントローラーが必要です。このアーキテクチャでは、ビデオ レートがビデオ コンポーネントの処理速度の影響を受けにくくなりますが、外部メモリに対するビデオ フレームの読み出しと書き込みに十分なメモリ帯域幅が求められます。

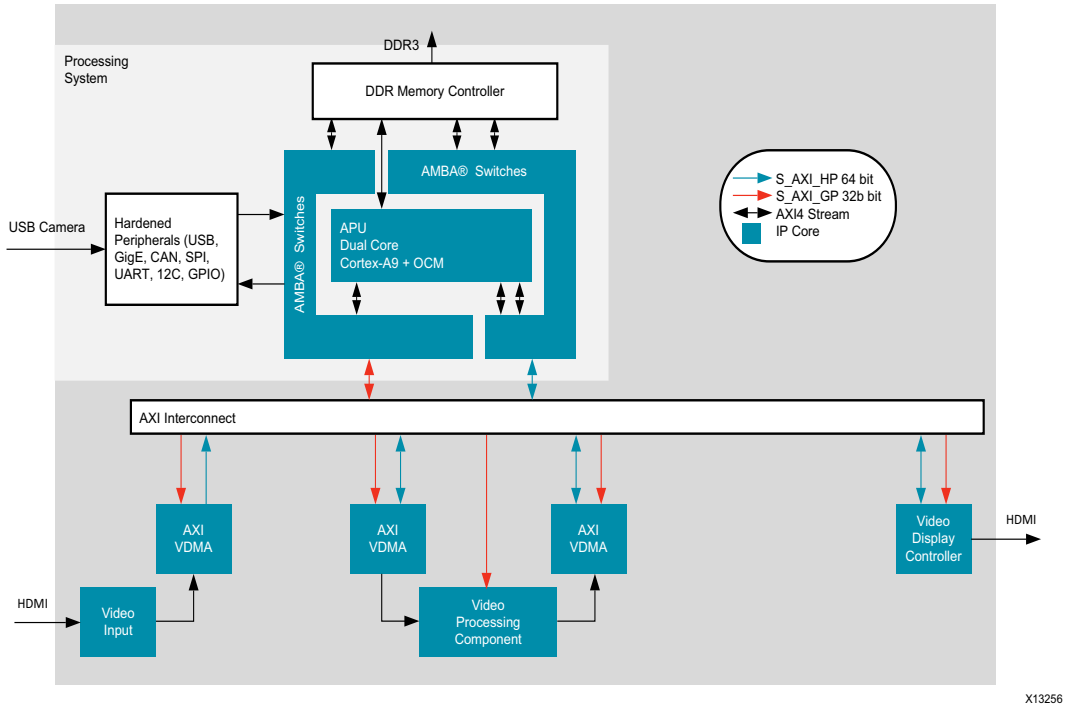

図 3 : ビデオ処理のフレーム バッファー アーキテクチャ

このアプリケーション ノートでは、柔軟性に優れ、プロセッサ コアでのビデオ処理を高速化する方法を理解しやすいフレーム バッファー ストリーミング アーキテクチャを扱います。高度に最適化されたシステムを実現するために、このアーキテクチャからダイレクト ストリーミング アーキテクチャを作成するのは比較的容易です。

AXI 4 ストリーミング ビデオ

ザイリンクスのビデオ処理コンポーネントは、一般的な AXI4 ストリーミング プロトコルを使用してピクセル データをやり取りします。このプロトコルは、ビデオ ピクセルの各ラインを AXI4 パケットとして記述し、各ラインの最後のピクセルは TLAST 信号のアサートでマークします。また、ビデオ フレームの開始は USER[0] ビットのアサートで最初のラインの先頭のピクセルをマークすることで示します。このプロトコルの詳細は、[参照 2] を参照してください。

基礎となる AXI4 ストリーミング ビデオ プロトコルには、イメージ内のライン サイズに対する制約はありませんが、すべてのビデオ ラインが同じ長さであれば、ほとんどの複雑なビデオ処理演算は大幅に簡略化されます。この制限は、一連のビデオ フレームの始め過渡状態の間を除けば、どのデジタル ビデオ フォーマットでもほぼ常に満たされます。この状態は通常、処理ブロックの入力インターフェイスでのみ問題となり、継続的な矩形フレームの処理に移行する前に適切に扱う必要があります。AXI4 ストリーミング ビデオ プロトコルを受信する入力インターフェイスでは、各ビデオ フレームが厳密に ROWS * COLS 個のピクセルで構成されるようにできます。そして、パイプライン内の後続のブロックは、ビデオ フレームが完全な矩形フレームであることを前提にできます。

Vivado HLS のビデオ インターフェイス ライブラリ

このようなインターフェイスの問題を扱いやすくするために、Vivado HLS には一連の合成可能なビデオ インターフェイス ライブラリ関数が含まれています。これらの関数を次の表に示します。

表 2 : Vivado HLS の合成可能なビデオ関数

ビデオ ライブラリ関数	説明
hls::AXIvideo2Mat	AXI4 ビデオ ストリーム表現のデータを hls::Mat フォーマットに変換する
hls::Mat2AXIvideo	hls::Mat フォーマットで格納されたデータを AXI4 ビデオ ストリームに変換する

具体的にいうと、AXIvideo2Mat 関数は AXI4 ストリーミング ビデオを使用する一連のイメージを受信し、hls::Mat 表現を生成します。同様に、Mat2AXIvideo 関数は一連のイメージの hls::Mat 表現を受信し、AXI4 ストリーミング ビデオ プロトコルを用いて正しくエンコードします。

これらの関数は、AXI4 ビデオ ストリームに基づいてイメージ サイズを決定するのではなく、hls::Mat コンストラクター引数で指定されたイメージ サイズを使用します。AXI4 ストリーミング インターフェイスで任意のサイズの入力イメージを処理するように設計されたシステムでは、ビデオ ライブラリ ブロックの外部でイメージ サイズを決定する必要があります。Zynq ビデオ TRD では、アクセラレータによって処理されるイメージ サイズは、AXI4-Lite 制御レジスタとしてアクセス可能です。ただし、ソフトウェアとシステムの残りの部分は、1920x1080 の解像度のみを処理するように設計されています。より複雑なシステムでは、ザイリンクスの Video Timing Controller コア [参照 3] を使用して、受信したビデオ信号のサイズを検出できます。

ビデオ ライブラリには、次の合成不可のビデオ インターフェイス ライブラリ関数も含まれています。

表 3 : Vivado HLS の合成不能なビデオ関数

ビデオ ライブラリ関数	
hls::cvMat2AXIvideo	hls::AXIvideo2cvMat
hls::IplImage2AXIvideo	hls::AXIvideo2IplImage
hls::CvMat2AXIvideo	hls::AXIvideo2CvMat

これらの関数は、通常は合成可能な関数と組み合わせて OpenCV ベースのテストベンチのインプリメントに使用されます。

制限

現在の合成可能なライブラリにはいくつかの制限があり、それらは注意しなければ見落としがちです。基本的な制限は、OpenCV 関数は直接合成できず、合成可能なライブラリの関数によって置き換えが必要なことです。この制限の理由は第一に、OpenCV 関数には通常は (合成可能ではない任意のサイズの `cv::Mat` オブジェクトのコンストラクターなど) 動的なメモリ割り当てが含まれるためです。

2 つ目の制限は、イメージのモデル化に使用される `hls::Mat<>` データ型は、外部メモリ内のピクセルの配列としてではなく、`hls::stream<>` データ型を使用して内部でピクセルのストリームとして定義されることです。その結果、イメージ上でランダム アクセスはサポートされません。また、`cv::Mat<>.at()` メソッドと `cvGet2D()` 関数に対応する等価な関数は合成可能なライブラリ内にありません。イメージへのアクセスはストリーミング アクセスになるため、複数の関数によってイメージを処理する場合は、`hls::Duplicate<>` 関数などで最初にイメージを 2 つのストリームに複製する必要があります。また、イメージのエリアを変更するには、イメージの周囲の未変更のピクセルを処理する必要があります。

もう 1 つの制限はデータ型に関するものです。OpenCV 関数は、通常は整数データ型または浮動小数点データ型をサポートします。しかし、プログラマブル ロジックをターゲットとする場合、浮動小数点型の方がよりコストがかかるため敬遠されます。ほとんどの場合、Vivado HLS で `float` 型および `double` 型を Vivado HLS の固定小数点テンプレート クラス `ap_fixed<>` および `ap_ufixed<>` に置き換えることができます。さらに、OpenCV と合成可能なライブラリの動作は概して機能的に同等になることことを意図していますが、OpenCV は浮動小数点演算を実行するため、合成可能なライブラリの結果がそれに対応する OpenCV 関数と等しいビット精度になることは想定できません。場合によっては、合成可能ライブラリは内部で固定小数点の最適化を実行し、浮動小数点演算を削減します。

最後の制限として、インターフェイス関数は AXI4 ストリーミング ビデオ用にのみ提供されます。これらのインターフェイスは、システム レベルでザイリンクスの Video DMA (VDMA) コアおよびほかのビデオ処理 IP コアと統合できますが、AXI4 スレーブ ポートや AXI4 マスター ポートに直接接続することはできません。したがって、(たとえば複数の連続するフレームをまとめて処理する目的で) 外部メモリ フレーム バッファーを必要とするデザインでは、プロセッサによって管理される複数の VDMA コアを使用して、フレーム バッファーを外部でインプリメントする必要があります。

リファレンス デザイン

このアプリケーション ノートには、複数の HLS デザインが含まれています。これらのデザインは、プログラマブル ロジック内の画像処理フィルターを Vivado HLS および Vivado HLS 合成可能ライブラリによって生成したフィルターで置き換え、Zynq ベース ターゲット リファレンス デザインの動作を変更します。合成可能なフィルターのゴールデン モデルが OpenCV ライブラリを使用してインプリメントされ、合成されたコードの動作が検証できるようになります。また、これらのデザインは Linux アプリケーションを変更し、フィルターの OpenCV インプリメンテーションまたは合成可能インプリメンテーションを Cortex-A9 コアで実行できるようにします。

リファレンス デザインには次のものがあります。

- demo : 一連のピクセル処理関数
- fast-corners : コーナー検出用の高速アルゴリズム
- pass-through : パススルー以外何も実行しない
- simple-median : 単純な Median フィルター
- simple-posterize : 単純なポスタライゼーション
- sobel : しきい値ありの Sobel フィルター (Zynq ベース TRD 内のデザインと同じ)

このセクションでは、3 つの標準的なリファレンス デザインを順に紹介し、OpenCV アプリケーションを高速化する方法の概要を示します。

最初のリファレンス デザインは「pass-through 」と呼ばれる透過的な画像フィルターで、入力フレームを変更せずに出力に渡すだけです。これはユーザーのアプリケーションでフィルターのテンプレートとして使用できます。OpenCV コードは次のように簡単です。

```
49 void opencv_image_filter(IplImage *src, IplImage *dst) {
50 cvCopy(src, dst);
51 }
```

OpenCV コードと比較すると、合成可能コードにはインターフェイスを容易にする #pragma 指示子が多数含まれています。これらの指示子は、入力および出力ストリームを AXI4 ストリーミング インターフェイスとしてアクセス可能にし (行 46-47)、バンドルされた制御バス内のその他の入力を AXI4 Lite スレーブ インターフェイスとしてアクセス可能にします (行 49-51)。TRD が使用していたドライバーとの位置を合わせるために、rows および cols のオフセットが指定されていることに注意してください。また、ブロックは同じサイズのイメージを繰り返し処理することが予想されるため、rows および cols 入力は安定入力として指定されます (行 53-54)。変数はパイプラインの各段を通してプッシュされる必要はないため、この指定によってさらに最適化できるようになります。最後に、dataflow モードが選択されることで (行 57)、ピクセルがブロック間でストリーミングされて、さまざまな処理関数の同時実行が可能になります。

```
44 void image_filter(AXI_STREAM& video_in, AXI_STREAM& video_out, int rows, int cols) {
45     //Create AXI streaming interfaces for the core
46 #pragma HLS INTERFACE axis port=video_in bundle=INPUT_STREAM
47 #pragma HLS INTERFACE axis port=video_out bundle=OUTPUT_STREAM
48
49 #pragma HLS INTERFACE s_axilite port=rows bundle=CONTROL_BUS offset=0x14
50 #pragma HLS INTERFACE s_axilite port=cols bundle=CONTROL_BUS offset=0x1C
51 #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
52
53 #pragma HLS INTERFACE ap_stable port=rows
54 #pragma HLS INTERFACE ap_stable port=cols
55
56    YUV_IMAGE img_0(rows, cols);
57 #pragma HLS dataflow
58    hls::AXIvideo2Mat(video_in, img_0);
59    hls::Mat2AXIvideo(img_0, video_out);
60 }
```

demo という名前の第 2 のリファレンス デザインには、図 4 に示す関数の簡単なパイプラインが含まれています。

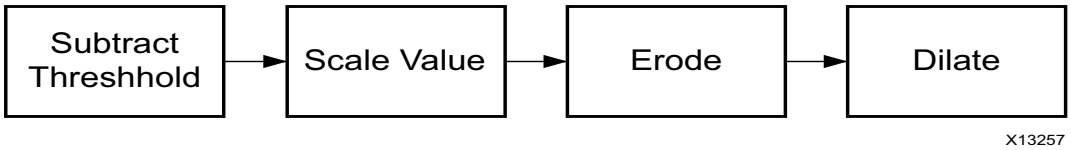

図 4 :「demo」のデザイン ブロック図

OpenCV では、次のコード例に示す一連のライブラリ呼び出しを使用して、このアプリケーションをインプリメントできます (apps/demo/opencv_top.cpp からの抜粋)。

```
49 void opencv_image_filter(IplImage *src, IplImage *dst) {
50 IplImage* tmp = cvCreateImage(cvGetSize(src), src->depth, src->nChannels);
51     cvCopy(src, tmp);
52     cvSubS(tmp, cvScalar(50, 50), dst);
53     cvScale(dst, tmp, 2, 0);
54     cvErode(tmp, dst);
55     cvDilate(dst, tmp);
56     cvCopy(tmp, dst);
57     cvReleaseImage(&tmp);
58 }
```

一連の OpenCV 関数呼び出しは、類似したインターフェイスを持ち等価な動作を行う HLS ビデオ ライブラリ呼び出しで置き換えられています。pass-through デザインのテンプレートに基づいた合成可能コードは簡単であり、#pragma 指示子は同じです。pass-through デザインとの主な違いは、いくつかの hls::Mat 宣言 (IMAGE_C2 は top.h で定義される 2 チャネル YUV 画像のタイプ) を追加し、HLS ライブラリ関数を使用した処理パイプラインを接続していることです。

```
44 void image_filter(AXI_STREAM& video_in, AXI_STREAM& video_out, int rows, int cols) {
45     //Create AXI streaming interfaces for the core
46 #pragma HLS INTERFACE axis port=video_in bundle=INPUT_STREAM
47 #pragma HLS INTERFACE axis port=video_out bundle=OUTPUT_STREAM
48
49 #pragma HLS INTERFACE s_axilite port=rows bundle=CONTROL_BUS offset=0x14
50 #pragma HLS INTERFACE s_axilite port=cols bundle=CONTROL_BUS offset=0x1C
51 #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
52
53 #pragma HLS INTERFACE ap_stable port=rows
54 #pragma HLS INTERFACE ap_stable port=cols
55
56     IMAGE_C2 img_0(rows, cols);
57     IMAGE_C2 img_1(rows, cols);
58     IMAGE_C2 img_2(rows, cols);
59     IMAGE_C2 img_3(rows, cols);
60     IMAGE_C2 img_4(rows, cols);
61     PIXEL_C2 pix(50, 50);
62 #pragma HLS dataflow
63     hls::AXIvideo2Mat(video_in, img_0);
64     hls::SubS(img_1, pix, img_1);
65     hls::Scale(img_1, img_2, 2, 0);
66     hls::Erode(img_2, img_3);
67     hls::Dilate(img_3, img_4);
68     hls::Mat2AXIvideo(img_4, video_out);
69 }
```

「fast-corners」という名前の第 3 のリファレンス デザインには、図 5 に示すように、より複雑なパイプラインが含まれています。

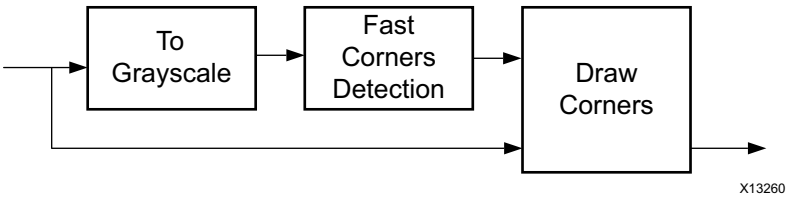

図 5 : Fast-Corners アプリケーション

OpenCV では、このアプリケーションは次のコード例を使用してインプリメントできます (apps/fast-corners/opencv_top.cpp からの抜粋)。

```
49 void opencv_image_filter(IplImage *_src, IplImage *_dst) {
50     Mat src(_src);
51     Mat dst(_dst);
52     cvCopy(_src, _dst);
53     std::vector<Mat> layers;
54     std::vector<KeyPoint> keypoints;
55     split(src, layers);
56     FAST(layers[0], keypoints, 20, true);
57     for (int i = 0; i < keypoints.size(); i++) {
58         rectangle(dst,
59                   Point(keypoints[i].pt.x-1, keypoints[i].pt.y-1),
60                   Point(keypoints[i].pt.x+1, keypoints[i].pt.y+1),
61                   Scalar(255,0),
62                   CV_FILLED);
63     }
64 }
```

このアプリケーションは簡単に見えますが、合成可能なライブラリ内に対応する関数を持たない関数がいくつか含まれています。std::vector クラスは合成できません (行 54)。cv::rectangle 関数は、置き換え更新のため、現在のところ合成可能ライブラリ内にインプリメントされていません (行 58-62)。

このアプリケーションをインプリメントする 1 つの可能性として、アプリケーションを分割し、プロセッシング システムで動作するコードによって FAST コーナーを矩形でマークしながら、プログラマブル ロジックでグリーン チャネルの抽出と FAST コーナーの検出を実行することが考えられます。ここでは、FPGA にインプリメントできるピクセル処理パラダイムの枠内に留まりながら、コードを若干簡略化することにします。次のコード例に、簡略化したコードを示します。

```
69 void opencv_image_filter(IplImage *_src, IplImage *_dst) {
70     Mat src(_src);
71     Mat dst(_dst);
72     cvCopy(_src, _dst);
73     Mat mask(src.rows, src.cols, CV_8UC1);
74     Mat dmask(src.rows, src.cols, CV_8UC1);
75     std::vector<Mat> layers;
76     std::vector<KeyPoint> keypoints;
77     split(src, layers);
78     FAST(layers[0], keypoints, 20, true);
79     GenMask(mask, keypoints);
80     dilate(mask, dmask, getStructuringElement(MORPH_RECT, Size(3,3), Point(1,1)));
81     PaintMask(dst, dmask, Scalar(255,0));
82 }
```

このコードは OpenCV で記述されていますが、合成可能コードへの変換が可能な形で構造化されています。この組み合わせは特に、動的に割り当てられた構造としてではなく、イメージ マスクとしてキーポイントを生成します。PaintMask 関数は、このようなマスクを使用してイメージの上に描画します。次のコード例に、FAST コーナー アプリケーションの合成可能バージョンを示します (apps/fast-corners/top.cpp からの抜粋)。

```
44 void image_filter(AXI_STREAM& video_in, AXI_STREAM& video_out, int rows, int cols) {
45     //Create AXI streaming interfaces for the core
46 #pragma HLS INTERFACE axis port=video_in bundle=INPUT_STREAM
47 #pragma HLS INTERFACE axis port=video_out bundle=OUTPUT_STREAM
```

```
48
49 #pragma HLS INTERFACE s_axilite port=rows bundle=CONTROL_BUS offset=0x14
50 #pragma HLS INTERFACE s_axilite port=cols bundle=CONTROL_BUS offset=0x1C
51 #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
52
53 #pragma HLS INTERFACE ap_stable port=rows
54 #pragma HLS INTERFACE ap_stable port=cols
55
56     IMAGE_C2 img_0(rows, cols);
57     IMAGE_C2 img_1(rows, cols);
58     IMAGE_C2 img_1_(rows, cols);
59     IMAGE_C1 img_1_Y(rows, cols);
60     IMAGE_C1 img_1_UV(rows, cols);
61     IMAGE_C2 img_2(rows, cols);
62     IMAGE_C1 mask(rows, cols);
63     IMAGE_C1 dmask(rows, cols);
64     PIXEL_C2 color(255,0);
65 #pragma HLS dataflow
66 #pragma HLS stream depth=20000 variable=img_1_.data_stream
67     hls::AXIvideo2Mat(video_in, img_0);
68     hls::Duplicate(img_0, img_1, img_1_);
69     hls::Split(img_1, img_1_Y, img_1_UV);
70     hls::Consume(img_1_UV);
71     hls::FASTX(img_1_Y, mask, 20, true);
72     hls::Dilate(mask, dmask);
73     hls::PaintMask(img_1_, dmask, img_2, color);
74     hls::Mat2AXIvideo(img_2, video_out);
75 }
```

このコードでは、行 66 の指示子の使用によって 1 つのストリームの深さが設定されていることに注意してください。この例では、ライン バッファーがあるため、`Duplicate > FASTX > Dilate > PaintMask` のパス上にビデオ ライン単位のレイテンシがあります。この場合、`FASTX` で最大 7 ライン、`Dilate` で最大 3 ラインのレイテンシが発生するため、少なくとも 1920 * (7+3) < 20000 ピクセルが必要です。さらに、行 71 の `hls::FASTX` 呼び出しは、前の変更された OpenCV コードの `cv::FAST` 関数と `GenMask` 関数の組み合わせに対応します。

ソース ファイルとプロジェクト ディレクトリ

このアプリケーション ノートのファイルの全体構造は、Zynq ベース TRD の構造を反映しています。SD カード用にあらかじめ作成されたイメージが ready_to_test ディレクトリに用意されており、デザインを簡単に試すことができます。このイメージは Zynq ベース TRD から若干変更されており、ブート時にロードされる RAM ディスク イメージに、ARM 用の OpenCV ライブラリのプリコンパイル済みバージョンが含まれています。デバイス ツリーも変更され、画像処理フィルター内の制御レジスタにアクセスするための汎用 UIO Linux カーネル ドライバーの適切なコンフィギュレーションが含まれています。apps ディレクトリには HLS リファレンス デザインが含まれています。デザインの再構築を簡単にするため、次のそれぞれをターゲットとする makefile も提供されています。

- make all -- sw および hw の作成、sd_image の生成
- make csim -- C シミュレーションの実行
- make cosim -- C/RTL 協調シミュレーションの実行
- make core -- 高位合成の実行と IP コアのエクスポート
- make bitstream -- ビットストリームの生成
- make boot -- ブート イメージの生成
- make elf -- ソフトウェア アプリケーションの作成
- make help -- ヘルプの出力

各デザイン ディレクトリ (apps/<design name>) で、Linux コマンド ラインまたは Windows の Vivado HLS コマンド プロンプトから make all を実行するとデザイン全体が再作成され、実行可能な SD カード イメージが生成されます。新しいアプリケーションを作成するには、最初の apps/ サブディレクトリの 1 つをコピーし、ソース コードを変更して適切な makefile ルールを実行します。FPGA デザインの全体構造は C コード ベースでは変更されないため、ソース コードの変更は生成される RTLのインターフェイスを変更しないものに限られます。あるいは、FPGA デザインでも対応する箇所を変更するのであれば、インターフェイスを変更できます。

OpenCV アプリケーションを高速化する手順

このセクションでは、OpenCV アプリケーションを高速化し、ボードでテストするウォークスルーを示します。OpenCV アプリケーションの例として、画像処理の簡単な 2D フィルターである erode (収縮処理) を考えます。ここで示すウォークスルーに従って画像処理アルゴリズムを置き換えることで、独自の OpenCV アプリケーションを簡単に作成できます。このアプリケーション ノートに付属する 2 つのリファレンス デザインも、この手順に従って利用できます。

この目的を達成するには、次の前提要件があります。

- ザイリンクスの Zynq-7000 SoC ZC702 評価キットまたは Zynq-7000 SoC ビデオおよび画像処理キット
- HDMI™ ポートまたは DVI ポート搭載 (HDMI/DVI ケーブルが必要)、解像度 1920x1080、60 フレーム レート表示対応のモニター
- Linux/Windows ホスト
- XAPP1167 パッケージ (このアプリケーション ノートに付属)
- Vivado Design Suite 2014.4 System Edition
- (オプション) HDMI ライブ ビデオ入力を有効にするための FMC-IMAGEON ボード

次に説明する手順の大半は、Linux コマンド ラインで示されます。Windows 用には、同じコマンドに対応する Vivado HLS コマンド プロンプトを適切に設定されたパスで起動するバッチ ファイルが提供されます。Vivado ツールがデフォルトの場所にインストールされていない場合は、インストール パスを反映するようにバッチ ファイルを変更する必要があります。

最初にパッケージの内容を展開し、ホーム ディレクトリとして使用します。

```
$ export VIDEO_HOME=/path/to/the/extracted/package/root
$ cd ${VIDEO_HOME}
$ ls
apps  doc  hardware  ready_to_test  software  xapp1167_windows.bat
```

注記 : FMC-IMAGEON ボードを使用する場合は、行 95 の apps/common/configure.mk を WITH_FMC := y に変更し、適切なデバイス ツリーを選択します。デフォルトは WITH_FMC := n です。

手順 1 : 新しいデザインを作成する

pass-through デザインは、画像処理デザインのテンプレートとして使用できます。新しい erode デザインを作成するには、apps ディレクトリ内の pass-through デザイン ディレクトリをコピーします。

```
$ cd ${VIDEO_HOME}/apps
$ cp -r pass-through erode
$ cd erode
```

ソース ファイル opencv_top.cpp を編集し、cvCopy 関数を cvErode 関数で置き換えます。

```
void opencv_image_filter(IplImage *src, IplImage *dst) {
   cvErode(src, dst);
}
```

ソース ファイル top.cpp を編集し、各インターフェイス関数の間に HLS ビデオ ライブラリの erode 関数 hls::Erode を追加します。

```
 44 void image_filter(AXI_STREAM& video_in, AXI_STREAM& video_out, int rows, int cols) {
```

```
45     //Create AXI streaming interfaces for the core
46 #pragma HLS INTERFACE axis port=video_in bundle=INPUT_STREAM
47 #pragma HLS INTERFACE axis port=video_out bundle=OUTPUT_STREAM
48
49 #pragma HLS INTERFACE s_axilite port=rows bundle=CONTROL_BUS offset=0x14
50 #pragma HLS INTERFACE s_axilite port=cols bundle=CONTROL_BUS offset=0x1C
51 #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
52
53 #pragma HLS INTERFACE ap_stable port=rows
54 #pragma HLS INTERFACE ap_stable port=cols
55
56     YUV_IMAGE img_0(rows, cols);
57     YUV_IMAGE img_1(rows, cols);
58 #pragma HLS dataflow
59     hls::AXIvideo2Mat(video_in, img_0);
60     hls::Erode(img_0, img_1);
61     hls::Mat2AXIvideo(img_1, video_out);
62 }
```

C シミュレーションを実行してアルゴリズムを検証します。

```
$ make csim
```

これにより、`hls::Erode` を実行して出力イメージを生成し、OpenCV 関数 `cvErode` によって生成されるゴールデン イメージと比較するテストが作成されます。「Test passed!」と表示されると、2 つのイメージがまったく同一であることがわかります。出力イメージを表示して、収縮処理の結果を検証することを推奨します。

手順 2 : ARM 用 OpenCV アプリケーションを作成する

ホスト上で ARM アプリケーションをクロスビルドするには、ARM GNU ツールがインストールされている必要があります。ARM GNU ツールは、ザイリンクスのソフトウェア開発キット (SDK) に含まれています。このデザインでは、次のコマンド ルールを実行して ARM アプリケーションを作成します。

```
$ make elf
```

ビルドが完了すると、ARM 実行ファイル `video_cmd` が `apps/erode/software/xsdk/video_cmd/bin/video_cmd` に作成されます。ARM アプリケーションには、プロセッサで OpenCV の erode を実行するオプションがあります。

手順 3 : Vivado HLS を実行して IP コアを作成する

この手順では、Vivado HLS を使用してビデオ ライブラリ関数を合成し、次の手順のために IP コアを作成します。次のコマンドを実行して先に進みます。

```
$ make core
```

注記 : ここでは手順を迅速化するために C/RTL 協調シミュレーションを省略しています。ただし、Vivado HLS デザインでは常に協調シミュレーションを実行することを推奨します。協調シミュレーションは、次のコマンドで実行できます。

```
$ make cosim
```

手順 4 : アクセラレータを使用して新しいシステムを構築する

次のコマンドは、新たに生成された IP コアをハードウェア プロジェクトにコピーし、FPGA インプリメンテーション フローを開始してビット ストリームを生成します。

```
$ make bitstream
```

次に、ビット ストリーム ファイルによって、SD カードのブート イメージが、プリコンパイル済み FSBL 実行ファイルおよびプリコンパイル済み U-Boot 実行ファイルと共に生成されます。

```
$ make boot
```

ブート イメージは、boot/BOOT.bin に置かれます。ここで、新しいデザインのハードウェアとソフトウェアの両方がボードでテストできます。最後に、次のコマンドを実行して、オンボード テスト用の sd_image を生成します。

```
$ make all
```

注記 : デザイン ディレクトリで `make all` を実行すると、手順 2 ～ 4 が実行されます。この処理が完了すると、使用可能な状態の SD カード イメージが ./sd_image に作成されます。

手順 5 : ボードでテストを行う

生成された sd_image/ 内のすべてのファイルとディレクトリを、SD カードのルート ディレクトリにコピーします。

ボードを次のようにセットアップします。

- HDMI または HDMI/DVI ケーブルを使用して、ZC702 ボードの HDMI 出力ポートにモニターを接続します。
- UART 通信用に、ZC702 ボードの USB UART と表示された Mini USB ポート J17 に USB Mini-B ケーブルを接続し、ホスト PC の空き USB ポートに USB Type-A ケーブル コネクタを接続します。
- ZC702 ボードに電源を接続します。
- (オプション) 1080p60 ビデオを出力するビデオ ソースを FMC_IMAGEON ボードの HDMI 入力ポートに接続し、ライブ入力を有効にします。
- (オプション) RJ45 イーサネット ケーブルを使用して、ZC702 ボードのイーサネット ポートをネットワークに接続します。

図 6 に示すようにスイッチが設定されていることを確認します。これにより、ZC702 ボードを SD カードからブートできます。

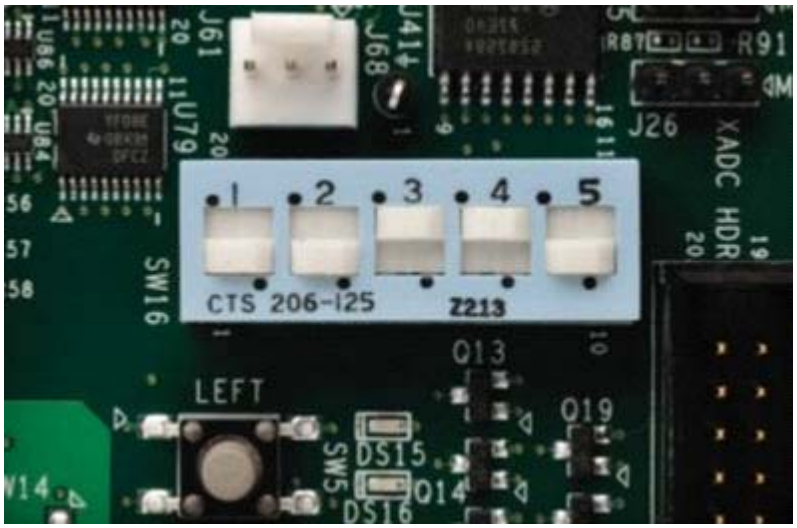

図 6 : ZC702 ボード上の SW16 スイッチの設定

端末プログラム (TeraTerm など) を開き、Baud Rate = 115200、Dat a bits = 8、Parity = None、Stop Bits = 1、Flow control = None に設定します。

sd_image の内容を格納した SD カードを、ZC702 ボードの SD スロットに挿入します。

ボードの電源をオンにして、システムが起動するのを待ち、root/root でログインします。

コマンド ライン モードでアプリケーションを実行します。

```
root@zynq:~# run_video.sh -cmd
```

まとめ

OpenCV は、コンピューター ビジョン デザインの開発に便利なフレームワークです。OpenCV アプリケーションは、ARM アーキテクチャ向けに再コンパイルして Zynq デバイスで実行することにより、エンベデッド システムでも使用可能になります。また、Vivado HLS の合成可能ビデオ ライブラリを利用して、OpenCV アプリケーションを高速化し、高精細ビデオをリアルタイムで処理できます。

参考資料

1. www.opencv.org
2. 『AXI リファレンス ガイド』(UG761)
3. 『Video Timing Controller 製品ガイド』(PG016)
4. 『Vivado Design Suite ユーザー ガイド : 高位合成』(UG902 : 英語版、日本語版)
5. 『Zynq ベース TRD』(UG925)
6. Vivado HLS のウェブ ページ : japan.xilinx.com/hls

改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	内容
2015 年 6 月 24 日	v3.0	最新リリースを反映するように「リファレンス デザイン」およびその他のセクションを更新。
2013 年 7 月 23 日	v2.0	最新リリースを反映するように各セクションを更新。
2013 年 3 月 20 日	v1.0	初版