XAPP1167 (v2.0) August 27, 2013

Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries

Author: Stephen Neuendorffer, Thomas Li, and Devin Wang

Summary

This application note describes how the OpenCV library can be used to develop computer vision applications on Zynq®-7000 All Programmable SoCs. OpenCV can be used at many different points in the design process, from algorithm prototyping to in-system execution. OpenCV code can also migrate to synthesizable C++ code using video libraries that are delivered with Vivado® High-Level Synthesis (HLS). When integrated into a Zynq SoC design, the synthesized blocks enable high resolution and frame rate computer vision algorithms to be implemented.

Introduction

Computer vision is a field that broadly includes many interesting applications, from industrial monitoring systems that detect improperly manufactured items to automotive systems that can drive cars. Many of these computer vision systems are implemented or prototyped using OpenCV, a library which contains optimized implementations of many common computer vision functions targeting desktop processors and GPUs. Although many functions in the OpenCV library have been heavily optimized to enable many computer vision applications to run close to real-time, an optimized embedded implementation is often preferable.

This application note presents a design flow enabling OpenCV programs to be retargeted to Zynq devices. The design flow leverages HLS technology in the Vivado Design Suite, along with optimized synthesizable video libraries. The libraries can be used directly, or combined with application-specific code to build a customized accelerator for a particular application. This flow can enable many computer vision algorithms to be quickly implemented with both high performance and low power. The flow also enables a designer to target high data rate pixel processing tasks to the programmable logic, while lower data rate frame-based processing tasks remain on the ARM® cores.

As shown in the Figure below, OpenCV can be used at multiple points during the design of a video processing system. On the left, an algorithm may be designed and implemented completely using OpenCV function calls, both to input and output images using file access functions and to process the images. Next, the algorithm may be implemented in an embedded system (such as the Zynq Base TRD), accessing input and output images using platform-specific function calls. In this case, the video processing is still implemented using OpenCV functions calls executing on a processor (such as the Cortex™-A9 processor cores in Zynq Processor System). Alternatively, the OpenCV function calls can be replaced by corresponding synthesizable functions from the Xilinx Vivado HLS video library. OpenCV function calls can then be used to access input and output images and to provide a golden reference implementation of a video processing algorithm. After synthesis, the processing block can be integrated into the Zynq Programmable Logic. Depending on the design implemented in the Programmable Logic, an integrated block may be able to process a video stream created by a processor, such as data read from a file, or a live real-time video stream from an external input.

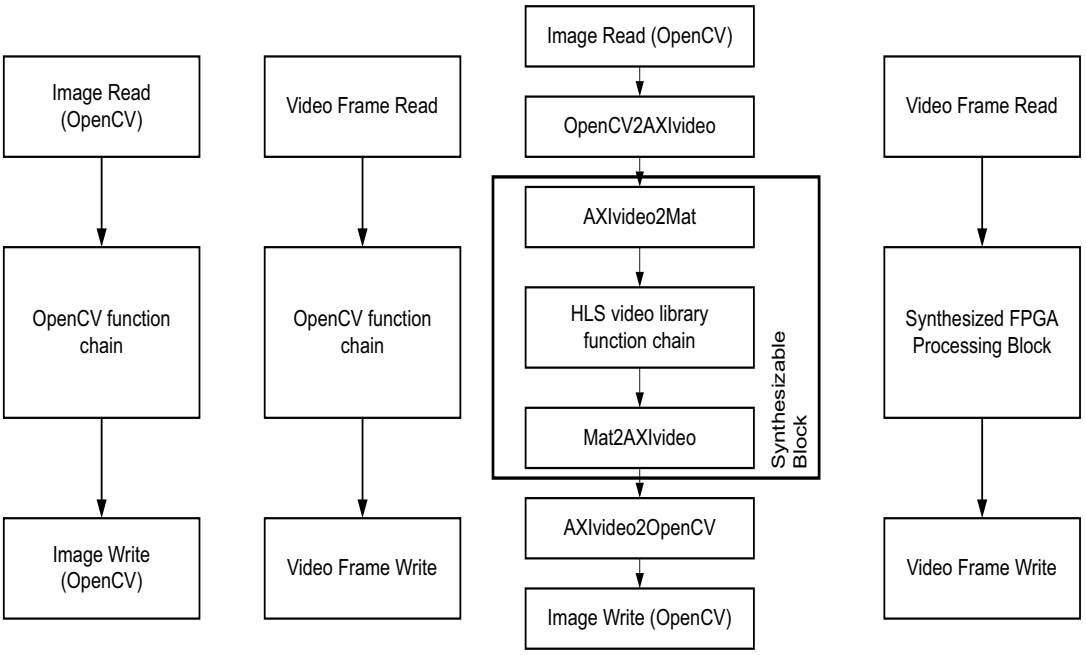

Figure 1: **Design Flow**

The design flow for this application note generally follows the steps below:

1. Develop and execute an OpenCV application on Desktop.
2. Recompile and execute the OpenCV application in the Zynq SoC without modification.
3. Refactor OpenCV application using I/O functions to encapsulate an accelerator function.
4. Replace OpenCV function calls with synthesizable video library function calls in accelerator function.
5. Generate an accelerator and corresponding API from the accelerator function using Vivado HLS.
6. Replace calls to the accelerator function with calls to the accelerator API
7. Recompile and execute the accelerated application

Reference Design

The reference design files can be downloaded from:

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

The reference design matrix is shown in Table 1.

Table 1: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Thomas Li
Target devices (stepping level, ES, production, speed grades)	XC7020-1
Source code provided	Yes
Source code format	C
Design uses code/IP from existing Xilinx application note/reference designs, CORE Generator software, or third party	Yes, based off the Zynq Base TRD application note/reference designs, CORE Generator software, or third party Simulation

Table 1: **Reference Design Matrix**

Parameter	Description
Simulation	
Functional simulation performed	Yes, in C
Timing simulation performed	No
Test bench used for functional and timing simulation	Provided
Test bench format	C
Simulation software/version used	g++
SPICE/IBIS simulations	No implementation
Implementation	
Synthesis software tools/version used	Vivado 2013.2
Implementation software tools/versions used	Vivado 2013.2
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	ZC702

Video Processing Libraries in Vivado HLS

Vivado HLS contains a number of video libraries, intended to make it easier for you to build a variety of video processing. These libraries are implemented as synthesizable C++ code and roughly correspond to video processing functions and data structures implemented in OpenCV. Many of the video concepts and abstractions are very similar to concepts and abstractions in OpenCV. In particular, many of the functions in the OpenCV imgproc module have corresponding Vivado HLS library functions.

For instance, one of the most central elements in OpenCV is the cv::Mat class, which is usually used to represent images in a video processing system. A cv::Mat object is usually declared something like:

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

This declares a variable **`image`** and initializes it to represent an image with 1080 rows and 1920 columns, where each pixel is represented by 3 eight bit unsigned numbers. The synthesizable library contains a corresponding hls::Mat<> template class that represents a similar concept in a synthesizable way:

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

The resulting object is similar, except the maximum size and format of the image are described using template parameters in addition to constructor arguments. This ensures that Vivado HLS can determine the size of memory to be used when processing this image and optimize the resulting circuit for a particular pixel representation. The hls::Mat<> template class also supports dropping the constructor arguments entirely when the actual size of the images being processed is the same as the maximum size:

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

Similarly, the OpenCV library provides a mechanism to apply a linear scaling to the value of each pixel in an image in the **`cvScale`** function. This function might be invoked like:

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

```
cvScale(src, dst, 2.0, 0.0);
```

This function call scales pixels in the input image **src** by a factor of 2 with no offset and generates an output image **dst**. The corresponding behavior in the synthesizable library is implemented by calling the **hls::Scale** template function:

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

1. Notice that although **hls::Scale** is a template function, the template arguments need not be specified, since they are inferred from the template arguments in the declaration of **src** and **dst**. The **hls::Scale** template function does require, however, that the input and output images have the *same* template arguments. A complete list of the supported functions can be found in the *Vivado Design Suite User Guide: High-Level Synthesis (*UG902*)* [4].

Architectures for Video Processing

Video Processing Designs in Zynq SoCs commonly follow one of the following two generic architectures. In the first architecture, referred to as “direct streaming”, pixel data arrives at the input pins of the Programmable Logic and is transmitted directly to a video processing component and then directly to a video output. A direct streaming architecture is typically the simplest and most efficient way to process video, but it requires that the video processing component be able to process frame strictly in real time.

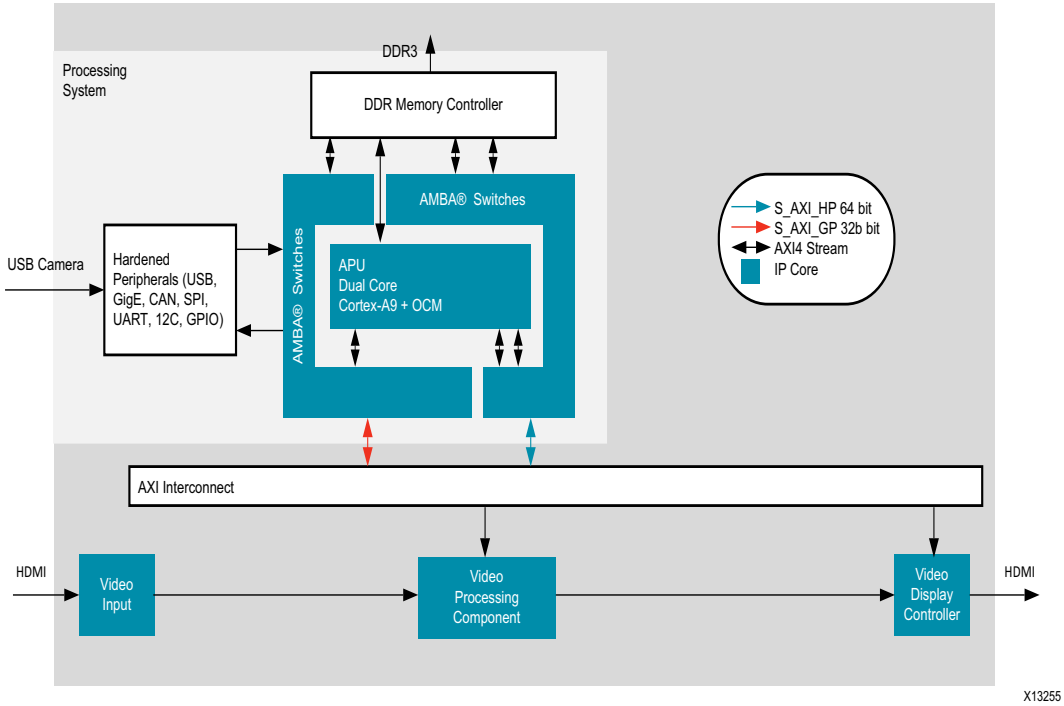

Figure 2: **Direct Streaming Architecture for Video Processing**

In the second architecture, refer to as “frame-buffer streaming”, pixel data is first stored in external memory before being processed and stored again in external memory. A video display controller is then required to output the processed video. A frame-buffer streaming architecture allows more decoupling between the video rate and the processing speed of the video component, but it requires enough memory bandwidth to read and write the video frames into external memory.

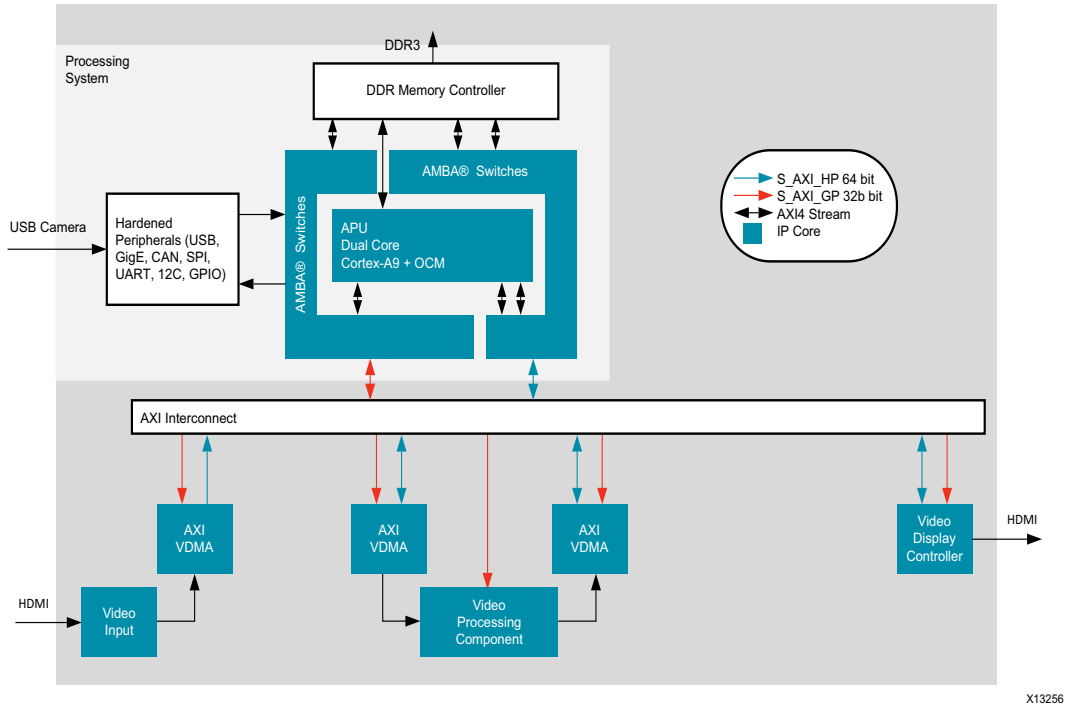

Figure 3: **Frame-buffer Architecture for Video Processing**

This application note focuses on the frame-buffer streaming architecture, since it provides more flexibility and is easier to understand how video processing on the processor cores can be accelerated. For highly optimized systems, it is relatively straightforward to construct a direct streaming architecture from a frame-buffer streaming architecture.

AXI 4 Streaming Video

Video Processing Components from Xilinx generally use a common AXI4 Streaming protocol to communicate pixel data. This protocol describes each line of video pixels as an AXI4 packet, with the last pixel of each line marked with the TLAST signal asserted. In addition, the start of the video frame is indicated by marking the first pixel of the first line with the USER[0] bit asserted. For more information about this protocol see [2].

Although the underlying AXI4 Streaming Video protocol does not require constraints on the size of lines in an image, most complex video processing computations are greatly simplified when all of the video lines are the same length. This restriction is almost always satisfied by any digital video format, except perhaps in a transient which occurs at the beginning of a sequence of video frames. Dealing with such transients is usually only a problem on the input interface of a processing block, which needs to correctly handle this transient before transitioning to processing continuous rectangular frames. The input interface that receives an AXI4 Streaming Video protocol can ensure that each video frame consists of exactly ROWS * COLS pixels. Then later blocks in the pipeline can assume that video frames are complete and rectangular.

Video Interface Libraries in Vivado HLS

To abstract a programmer from these interfacing issues, Vivado HLS includes a set of synthesizable video interface library functions. These functions are shown in the table below.

Table 2: **Vivado HLS Synthesizable Video Functions**

Video Library Function	Description
hls::AXIvideo2Mat	Converts data from an AXI4 video stream representation to hls::Mat format.
hls::Mat2AXIvideo	Converts data stored as hls::Mat format to an AXI4 video stream.

In particular, the AXIvideo2Mat function receives a sequence of images using the AXI4 Streaming Video and produces an hls::Mat representation. Similarly, the Mat2AXIvideo function receives an hls::Mat representation of a sequence of images and encodes it correctly using the AXI4 Streaming video protocol.

These functions don't determine the image size based on AXI4 video stream, but use the image size specified in the hls::Mat constructor arguments. In systems designed to process an arbitrary size input image with AXI4 Streaming interfaces, the image size must be determined externally to the video library block. In the Zynq Video TRD, the image size processed by the accelerator is exposed as AXI4-Lite control registers, however the software and the rest of the system is only designed to process 1920x1080 resolutions. In more complicated systems, the Xilinx Video Timing Controller core [3] could be used to detect the size of a received video signal.

The video libraries also contain the following non-synthesizable video interface library functions:

Table 3: **Vivado HLS Non-Synthesizable Video Functions**

Video Library Functions	
hls::cvMat2AXIvideo	hls::AXIvideo2cvMat
hls::IplImage2AXIvideo	hls::AXIvideo2IplImage
hls::CvMat2AXIvideo	hls::AXIvideo2CvMat

These functions are commonly used in conjunction with the synthesizable functions to implement OpenCV-based testbenches.

Colorspaces

One important thing to observe is that most OpenCV functions are colorspace independent, meaning that they affect each component of a pixel equally. There are, however, a few notable exceptions, where the byte ordering must be specified, and a few cases, where a default byte order is used. This default byte order (called BGR or BGRA), places the "B" component in the low-order bits and the R or A component in the high-order bits of a 32-bit word. The AXI4 Streaming Video protocol, however, specifies a particular (somewhat unusual) byte order, which is not directly supported by OpenCV. This format places the "G" component in the low-order bits, followed by the "B" and "R" components.

In the Zynq Video TRD, a portion of the design uses the AXI4 Streaming Video byte order, but frames in memory are stored in the more common BGR byte order. As a result, the example code in this application note uses the BGR byte order, which enables consistently using OpenCV as a design testbench.

Limitations

There are several limitations to the current synthesizable library, which may not be otherwise obvious. The basic limitation is that OpenCV functions cannot be synthesized directly, and must be replaced by functions from the synthesizable library. This limitation is primarily because OpenCV functions typically include dynamic memory allocation, such as during the constructor of an arbitrarily sized `cv::Mat` object, which is not synthesizable.

A second limitation is that the **hls::Mat<>** datatype used to model images is internally defined as a stream of pixels, using the **hls::stream<>** datatype, rather than as an array of pixels in external memory. As a result, random access is not supported on images, and the **cv::Mat<>.at()** method and **cvGet2D()** function have no correspondence. Streaming access also implies that if an image is processed by more than one function, then it must first be duplicated into two streams, such as by using the **hls::Duplicate<>** function. Streaming access also implies that an area of an image cannot be modified without processing the unmodified pixels around the image.

Another limitation relates to datatypes. OpenCV functions typically support either integer or floating-point datatypes. However, floating point is often more expensive and avoided when targeting programmable logic. In many cases, Vivado HLS supports replacing **float** and **double** types with the Vivado HLS fixed point template classes **ap_fixed<>** and **ap_ufixed<>**. Currently this is not uniformly supported in the synthesizable libraries, since certain functions (such as **hls::Filter2D** and **hls::ConvertScale**) require floating-point arguments. Additionally, since OpenCV performs floating-point operations, it should not be expected that the results of the synthesizable library are generally bit-accurate to the corresponding OpenCV functions, although the intention is generally for the behavior to be functionally equivalent. In some cases the synthesizable libraries do perform internal fixed-point optimizations to reduce the use of floating-point operations.

A final limitation is that interface functions are only provided for AXI4 Streaming Video. These interfaces can be integrated at the system level with the Xilinx Video DMA (VDMA) core and other video processing IP cores, but cannot be directly connected to AXI4 Slave ports or AXI4 Master ports. One implication of this is that in designs that require external memory frame buffers (for instance, in order to process several consecutive frames together) the frame buffer must be implemented externally with several VDMA cores managed by the processor.

Reference Designs

This app note contains two HLS designs. These designs modify the behavior of the Zynq Base Targeted Reference Design, replacing the image processing filter in the programmable logic with a filter generated using Vivado HLS and the Vivado HLS synthesizable libraries. The golden model of the synthesizable filter is implemented using OpenCV libraries, enabling the behavior of the synthesized code to be verified. The designs also modify the Linux application, enabling either an OpenCV implementation of the filter or the synthesizable implementation of the filter to be executed on Cortex-A9 cores. In addition, the app note is also packaged with minimal precompiled library binaries of OpenCV for both ARM target and x86 host architectures.

The first reference design, named “demo”, contains a simple pipeline of functions shown in Figure 4.

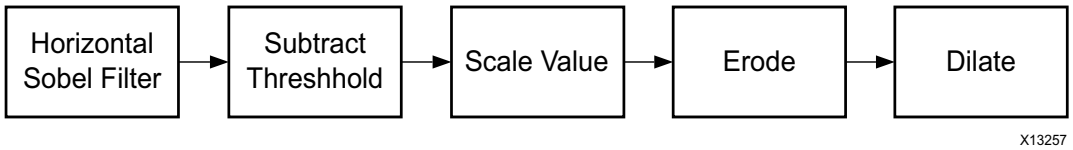

Figure 4: **"Demo" Design Block Diagram**

In OpenCV, this application can be implemented using a sequence of library calls, as shown in the following code example, excerpted from `sw/demo/opencv_top.cpp`.

```
1:void opencv_image_filter(
2:    IplImage* src, IplImage* dst) {
3:  cvSobel(src, dst, 1, 0);
4:  cvSubS(dst, cvScalar(50,50,50), src);
5:  cvScale(src, dst, 2, 0);
6:  cvErode(dst, src);
7:  cvDilate(src, dst);
```

```
8:}
```

The synthesizable code also contains a number of **#pragma** directives that enable ease of interfacing. These directives expose the input and output streams as AXI 4 Streaming interfaces (lines 26-29) and expose the other inputs and the control bus as an AXI 4 Lite Slave interface (lines 30-35). In addition, the **rows** and **cols** inputs are specified as being stable inputs, since the block is expected to process the same size images repeatedly (lines 36-37). This specification enables additional optimization on these variables, since they need not be pushed through each level of the pipeline. Lastly *dataflow* mode is selected (line 45), enabling concurrent execution of the various processing functions, with pixels being streamed from one block to another.

```
4:
5:void Sobel(RGB_IMAGE& src, RGB_IMAGE& dst, int dx, int dy) {
6:  // special case of sobel filter.
7:  assert(dx == 1 && dy == 0);
8:  const int k_val[3][3] =
9:       { {-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}, };
10:  hls::Window<3, 3, int> kernel;
11:  hls::Point_<int> anchor;
12:  for (int i = 0; i < 3; i++) {
13:  #pragma HLS unroll
14:    for (int j = 0; j < 3; j++) {
15:    #pragma HLS unroll
16:       kernel.val[i][j] = k_val[i][j];
17:    }
18:  }
19:  anchor.x = -1;
20:  anchor.y = -1;
21:  hls::Filter2D(src, dst, kernel, anchor);
22:}
23:void image_filter(AXI_STREAM& input, AXI_STREAM& output,
24:                  int rows, int cols) {
25:  //Create AXI streaming interfaces for the core
26:  #pragma HLS RESOURCE variable=input core=AXIS
27:                       metadata="-bus_bundle INPUT_STREAM"
28:  #pragma HLS RESOURCE variable=output core=AXIS
29:                       metadata="-bus_bundle OUTPUT_STREAM"
30:  #pragma HLS RESOURCE core=AXI_SLAVE variable=rows
31:                       metadata="-bus_bundle CONTROL_BUS"
32:  #pragma HLS RESOURCE core=AXI_SLAVE variable=cols
33:                       metadata="-bus_bundle CONTROL_BUS"
34:  #pragma HLS RESOURCE core=AXI_SLAVE variable=return
35:                       metadata="-bus_bundle CONTROL_BUS"
36:  #pragma HLS INTERFACE ap_stable port=rows
37:  #pragma HLS INTERFACE ap_stable port=cols
38:  RGB_IMAGE img_0(rows, cols);
39:  RGB_IMAGE img_1(rows, cols);
40:  RGB_IMAGE img_2(rows, cols);
41:  RGB_IMAGE img_3(rows, cols);
42:  RGB_IMAGE img_4(rows, cols);
43:  RGB_IMAGE img_5(rows, cols);
44:  RGB_PIXEL pix(50, 50, 50);
45:  #pragma HLS dataflow
46:  hls::AXIvideo2Mat(input, img_0);
47:  Sobel(img_0, img_1, 1, 0);
48:  hls::SubS(img_1, pix, img_2);
49:  hls::Scale(img_2, img_3, 2, 0);
50:  hls::Erode(img_3, img_4);
51:  hls::Dilate(img_4, img_5);
52:  hls::Mat2AXIvideo(img_5, output);
```

```
53:}
```

The second reference design, named “fast-corners”, contains a more complex pipeline, shown in Figure 5.

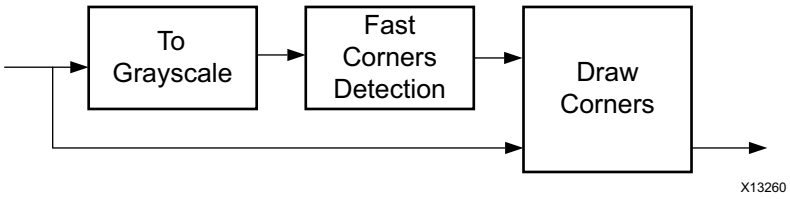

Figure 5: **Fast-Corners Application**

In OpenCV, this can be implemented using the following code example, excerpted from `sw/fast-corners/opencv_top.cpp`.

```
void opencv_image_filter(IplImage* src, IplImage* dst) {
    IplImage* gray = cvCreateImage( cvGetSize(src), 8, 1 );
    std::vector<cv::KeyPoint> keypoints;
    cv::Mat gray_mat(gray,0);

    cvCvtColor( src, gray, CV_BGR2GRAY );
    cv::FAST( gray_mat, keypoints, 20,true );
    cvCopy( src,dst);

    for (int i=0;i<keypoints.size();i++) {
      cvRectangle(dst,
                  cvPoint(keypoints[i].pt.x-1,keypoints[i].pt.y-1),
                  cvPoint(keypoints[i].pt.x+1,keypoints[i].pt.y+1),
                  cvScalar(255,0,0),
                  CV_FILLED);
    }

    cvReleaseImage( &gray );
}
```

Although it looks simple, this application contains quite a few functions that do not have corresponding functions in the synthesizable library. The **`cvCreateImage`** and **`cvReleaseImage`** functions represent dynamic memory allocation which is not synthesizable (lines 2-3 and 17). The **`std::vector`** class is not synthesizable (line 6). The **`cv::FAST`** and **`cvRectangle`** functions are not currently implemented in the synthesizable library (line 8 and 12-15).

One possibility for implementing this application is to partition it, running the green channel extraction and FAST corner detection in programmable logic, while marking the corners with rectangles with code running on the processing system. Instead we choose to simplify the code slightly, while staying within the pixel processing paradigm that can be implemented in the FPGA. The simplified version of the code is shown in the following code example.

```
1:void fast_corner(IplImage* img, IplImage* dst ) {
2:  IplImage* gray = cvCreateImage(
3:      cvSize(img->width,img->height), 8, 1 );
4:  cvCvtColor( img, gray, CV_BGR2GRAY );
5:  std::vector<cv::KeyPoint> keypoints;
6:  cv::Mat gray_mat(gray,0);
7:  cv::FAST(gray_mat, keypoints, 20,true );
8:  int rect=2;
9:  cvCopy(img,dst);
10:  for (int i=0; i<keypoints.size(); i++) {
11:     cvRectangle(dst,
```

```
12:       cvPoint(keypoints[i].pt.x,keypoints[i].pt.y),
13:      cvPoint(keypoints[i].pt.x+rect,keypoints[i].pt.y+rect),
14:      cvScalar(255,0,0),1);
15:  }
16:  cvReleaseImage( &gray );
17:}
```

This code, although written in OpenCV is structured in a way that is amenable to transformation into synthesizable code. The significant missing functionality is the combination of the **cv::FAST** function and the **GenMask** function, which can be implemented using application specific synthesizable code. In particular, this combination generates the keypoints as an image mask, rather than as a dynamically allocated structure. The **PrintMask** function takes such as mask and draws on top of the image.

The synthesizable version of the fast corners application is shown in the following code example, excerpted from `fast-corners/top.cpp`. The details of the synthesizable implementation of the FAST algorithm can be found in `fast-corners/hls_video_fast.h`.

```
1:void image_filter(AXI_STREAM& input, AXI_STREAM& output,
2:                  int rows, int cols) {
3:  //Create AXI streaming interfaces for the core
4:  #pragma HLS RESOURCE variable=input core=AXIS
5:          metadata="-bus_bundle INPUT_STREAM"
6:  #pragma HLS RESOURCE variable=output core=AXIS
7:          metadata="-bus_bundle OUTPUT_STREAM"
8:  #pragma HLS RESOURCE core=AXI_SLAVE variable=rows
9:          metadata="-bus_bundle CONTROL_BUS"
10:  #pragma HLS RESOURCE core=AXI_SLAVE variable=cols
11:          metadata="-bus_bundle CONTROL_BUS"
12:  #pragma HLS RESOURCE core=AXI_SLAVE variable=return
13:          metadata="-bus_bundle CONTROL_BUS"
14:  #pragma HLS interface ap_stable port=rows
15:  #pragma HLS interface ap_stable port=cols
16:  hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _src(rows,cols);
17:  hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _dst(rows,cols);
18:  #pragma HLS dataflow
19:  hls::AXIvideo2Mat(input, _src);
20:  hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      src0(rows,cols);
21:  hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      src1(rows,cols);
22:  #pragma HLS stream depth=20000 variable=src1.data_stream
23:  hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      mask(rows,cols);
24:  hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      dmask(rows,cols);
25:  hls::Scalar<3, unsigned char> color(255,0,0);
26:  hls::Duplicate(_src,src0,src1);
27:  FASTX(src0,mask,20,true);
28:  hls::Dilate(mask,dmask);
29:  PaintMask(src1,dmask,_dst,color);
30:  hls::Mat2AXIvideo(_dst, output);
31:}
```

One important thing to observe about this code is the use of the directive in line 22 to set the depth of one of the streams. In this case, there is a latency of several video lines on the path from Duplicate > FAST > Dilate > PaintMask because of the presence of line buffers. In this case, FAST incurs up to 7 lines of latency, Dilate incurs up to 3 lines of latency, which implies a need of at least: 1920 * (7+3) < 20000 pixels. In addition, the FASTX call in line 27 corresponds to the combination of the **cvCvtColor**, **cv::FAST**, and **GenMask** functions in Figure 6.

Source Files and Project Directories

The overall structure of the files in the application note mirrors the structure of the Zynq Base TRD. A prebuilt image for an SD card is provided in `sd_image` directory, making it easy to try the design out. This image is slightly modified from the Zynq Base TRD to include a precompiled version of the OpenCV libraries for ARM in a separate ramdisk image that is loaded at boot time. The device tree has also been modified to include the correct configuration of the generic-uio Linux kernel driver for accessing control registers in the image processing filter. The `sw` directory contains two packages for the 'demo' application and for the 'fast-corners' design. To enable easily rebuilding the designs, a Makefile is also provided with the following targets:

- make elf -- build sw (video_library_cmd)
- make sim -- build and run csim test
- make core -- synthesis and export IP core
- make bitstream -- generate bistream (system.bit)
- make boot -- generate boot image
- make all -- build sw and hw, generate sd_image

Running 'make all' from the Linux command line or the Vivado HLS command prompt under Windows rebuilds the entire design and generates an SD card image that can be run. New applications can be made by copying one of the initial `sw/` subdirectories, modifying the source code, and running the appropriate makefile rules. The overall structure of the FPGA design is not modified based on the C code, so care must be taken to limit modifications to those that do not modify the interface of the generated RTL. Alternatively, the interface can be modified, as long the corresponding changes are made in the FPGA design.

Steps to Accelerate an OpenCV Application

In this section we will demonstrate a walkthrough of how to accelerate an OpenCV application and test it on board. As an example, the OpenCV application is a simple 2D filter of image processing: erode. By following this walkthrough, one can easily write their OpenCV application by replacing the image processing algorithm, and the two reference designs included in the package are also applicable to follow this instruction.

To achieve this goal, several prerequisites are needed:

- Xilinx Zynq-7000 SoC ZC702 Evaluation Kit or Xilinx Zynq-7000 SoC Video and Imaging Kit
- Monitor with HDMI™ port or DVI port (HDMI/DVI cable needed), supports 1920x1080 resolution, 60 frame rate display
- Linux/Windows host
- Video library package (ship with this app note)
- ARM GNU tools
- Vivado System Edition
- ISE® Design Suite 14.6 (if using EDK-based design)

Most of the following steps will be demonstrated in Linux command line. For Windows, a batch file that starts the Vivado HLS command prompt with correctly set paths is provided that works with the same commands. This batch file may need to be modified to reflect the installation path of the ISE and Vivado tools if they are not installed in the default locations.

First, extract the content of the package, use it as home directory:

```
$ export VIDEO_HOME=/path/to/the/extracted/package/root
$ cd ${VIDEO_HOME}
$ ls
boot_image  hw  opencv_install  README.txt  sd_image  sw
```

Step 1. Create new design

The 'demo' design includes the basic structure for combinations of image processing functions. To create the new erode design, copy the `demo` design directory in `sw` directory:

```
$ cd ${VIDEO_HOME}/sw
$ cp –r demo erode
$ cd erode
```

Edit the source file `opencv_top.cpp`, modify the 5 pipeline functions to just single erode function:

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

Edit the source file `top.cpp` in the same way:

```
void image_filter(AXI_STREAM& input,
                  AXI_STREAM& output,
                  int rows,
                  int cols) {
//Create AXI streaming interfaces for the core
#pragma HLS RESOURCE variable=input core=AXIS
metadata="-bus_bundle INPUT_STREAM"
#pragma HLS RESOURCE variable=output core=AXIS
        metadata="-bus_bundle OUTPUT_STREAM"
#pragma HLS RESOURCE core=AXI_SLAVE variable=rows
             metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=cols
             metadata="-bus_bundle CONTROL_BUS"
#pragma HLS RESOURCE core=AXI_SLAVE variable=return
            metadata="-bus_bundle CONTROL_BUS"
#pragma HLS INTERFACE ap_stable port=rows
#pragma HLS INTERFACE ap_stable port=cols
   RGB_IMAGE img_0(rows, cols);
   RGB_IMAGE img_1(rows, cols);
   hls::AXIvideo2Mat(input, img_0);
   hls::Erode(img_0, img_1);
   hls::Mat2AXIvideo(img_1, output);
}
```

Note: In the new `top.cpp`, we use the corresponding function in **`hls namespace`** to erode an image. Run C simulation to verify the algorithm:

```
$ make sim
```

It will build a test to run **`hls::Erode`** to generate an output image to compare with the golden image generated by OpenCV function **`cvErode`**. See "Test passed!" to verify the two images are exactly the same. It is also recommended to view the output images to verify the result of erode.

Step 2. Build OpenCV application for ARM

To cross-build ARM applications on host, the ARM GNU tools must be installed. The ARM GNU tools are included with the Xilinx Software Development Kit (SDK). The opencv_install directory also includes the precompiled version of OpenCV library for ARM at ${VIDEO_HOME}/opencv_install/arm. For more information on using ARM GNU tools or building OpenCV library, please refer to http://opencv.org/.

For this design, run the following make rule to build the ARM application:

```
$ make elf
```

Once the build is done, the ARM executable video_library_cmd is in the current directory. The ARM application has options to run OpenCV erode and HLS video library erode on processor.

Step 3. Run Vivado HLS to create an IPcore

This step will use Vivado HLS to synthesize the video library functions, then create an IP core for the next step. Run the following command to proceed.

```
$ make core
```

Note: In order to hasten this step, C/RTL co-simulation is omitted here. You can run co-simulation by uncommenting the line with 'cosim_design' in run.tcl then re-run Vivado HLS:

```
$ vivado_hls run.tcl
```

Note: Running C/RTL co-simulation is always a recommended step in a Vivado HLS design.

Step 4. Build new system with the accelerator

The following command will copy the newly generated IP core to the hw project, run the FPGA implementation flow to generate a bit stream:

```
$ make bitstream
```

Next, the boot image of SD card will be generated by the bit stream file with the pre-compiled FSBL executable and the pre-compiled U-boot executable:

```
$ make boot
```

The boot image is at: ./boot_image/BOOT.bin. At this point, both hardware and software of the new design are ready to test on board.

Note: You can run '`make all`' in design directory to go through Step 2-4. Once it is done, the ready-for-use SD card image will be ready at ./sd_image.

Step 5. Test on board

In directory ${VIDEO_HOME}/sd_image, replace the old ARM executable video_library_cmd and boot image BOOT.bin with the newly generated ones, then copy all the files in sd_image/ to SD card.

Board setup:

- Connect the monitor to the HDMI out port of the ZC702 board using an HDMI or HDMI/DVI cable.
- Connect a USB Mini-B cable into the Mini USB port J17 labeled USB UART on the ZC702 board and the USB Type-A cable end into an open USB port on the host PC for UART communications.
- Connect the power supply to the ZC702 board.
- (Optional) Connect the video source which output 1080p60 video to the HDMI in port on FMC_IMAGE_ON board to enable live inputs.
- (Optional) Connect the Ethernet port on the ZC702 board to network using an RJ45 Ethernet cable.

Make sure the switches are set as shown in Figure 6, which allows the ZC702 board to boot from the SD card:

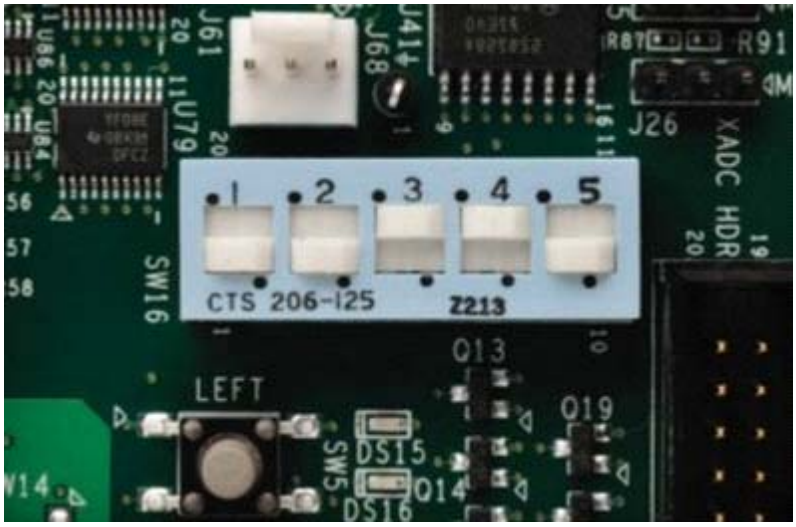

Figure 6: **Switches Setting of SW16 on ZC702 Board**

Open a terminal program (e.g. TeraTerm), set Baud Rate = 115200, Date bits = 8, Parity = None, Stop Bits = 1, and Flow control = None.

Insert the SD card which contains sd_image contents to the SD slot on the ZC702 board.

Switch the board power on, wait for system brought up login with root/root.

Run the application in command line mode:

```
# cd /mnt
# ./video_library_cmd
```

Conclusion

OpenCV is a useful framework for developing computer vision designs. OpenCV applications can be also used in embedded systems by recompiling them for the ARM architecture and executing them in Zynq devices. Additionally, by leveraging the synthesizable video libraries in Vivado HLS, OpenCV applications can be accelerated to process high-definition video in real-time.

Reference

1. www.opencv.org
2. AXI Reference Guide (UG761)
3. pg016 Video Timing Controller
4. Vivado Design Suite User Guide: High-Level Synthesis (UG902)
5. Zynq Base TRD (UG925)
6. Vivado HLS web page: www.xilinx.com/hls

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
3/20/13	v1.0	Initial Xilinx release.
7/23/13	v2.0	Updated sections to reflect current release.