Xilinx OpenCV HLS usemodel guide

Table of Contents

Using xfOpenCV in HLS

The xfOpenCV library can be used to build applications in Vivado HLS. In this context, this document provides details on how the xfOpenCV library components can be integrated into a design in Vivado HLS. This section of the document provides steps on how to run a single library component through the Vivado HLS use flow which includes, C-simulation, C-synthesis, C/RTL cosimulation and exporting the RTL as an IP.

The user needs to make the following changes to facilitate proper functioning of the use model in Vivado HLS.

1. Define xf::Mat objects as static in the testbench

In order to use xfOpenCV functions in standalone HLS mode, all the xf::Mat objects in the **testbench** must be declared with **static** keyword.

```
Ex: static xf::Mat<TYPE, HEIGHT, WIDTH, NPC1>
imgInput(in_img.rows,in_img.cols);
```

2. Use of appropriate compile-time options

When using the xfOpenCV functions in HLS, the following options need to be provided at the time of compilation : **__XFCV_HLS_MODE__, -std=c++0x**

3. Specifying interface pragmas to the interface level arguments

For the functions with top level interface arguments as pointers (with more than one read/write access) must have **m_axi** Interface pragma specified.

Ex:
```
void lut_accel(xf::Mat<TYPE, HEIGHT, WIDTH, NPC1> &imgInput,
xf::Mat<TYPE, HEIGHT, WIDTH, NPC1> &imgOutput, unsigned char *lut_ptr)
{
#pragma HLS INTERFACE m_axi depth=256 port=lut_ptr offset=direct
bundle=lut_ptr
      xf::LUT< TYPE, HEIGHT, WIDTH, NPC1>(imgInput,imgOutput,lut_ptr);
}
```

Operating HLS Standalone mode

HLS standalone mode can be operated in two modes:

1. using tcl script
2. using GUI

On command line with tcl script:

In the Vivado HLS tcl script file, update the following in the **cflags** of all **add_files** sections

1. Append the path to the **xfOpenCV/include** directory as it contains all the header files required by the library.

2. Add compiler flags **__XFCV_HLS_MODE__ , -std=c++0x**.
   - **__XFCV_HLS_MODE__** is to enable the static allocation of memory for xf::mat to support synthesis and co-sim.
   - **-std=c++0x** is to support the feature of setting default template parameter values.

Example:

Setting flags for source files:

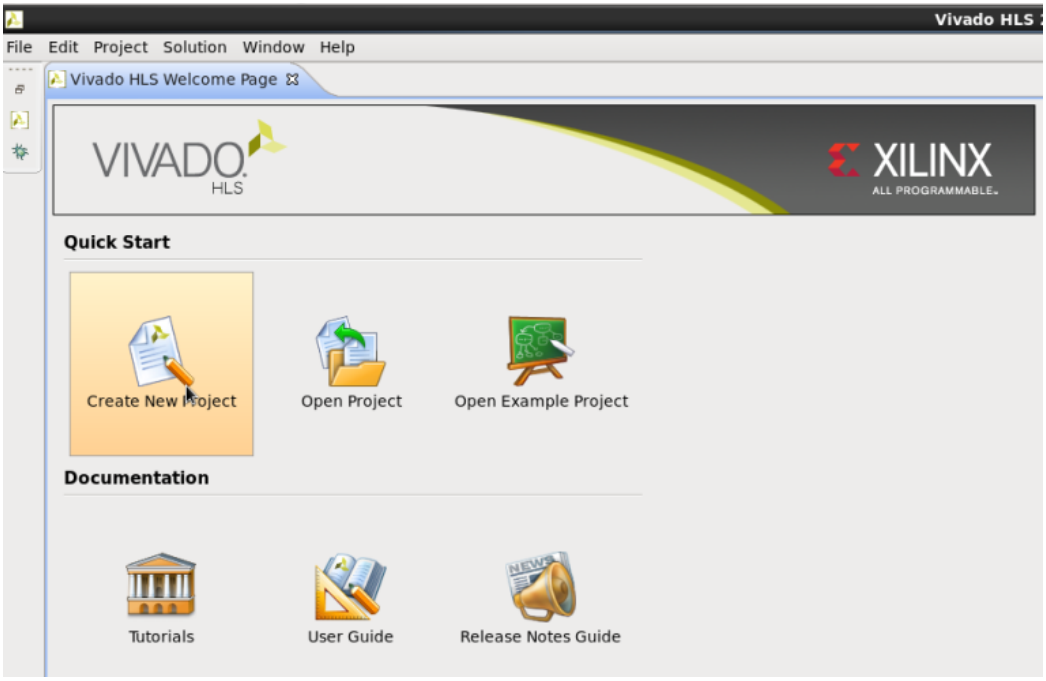
```
add_files xf_dilation_accel.cpp -cflags "-I<path-to-include-directory>
-D__XFCV_HLS_MODE__ -std=c++0x"
```

Setting flags for testbench files:

```
add_files -tb xf_dilation_tb.cpp -cflags "-I<path-to-include-directory>
-D __XFCV_HLS_MODE__ -std=c++0x"
```

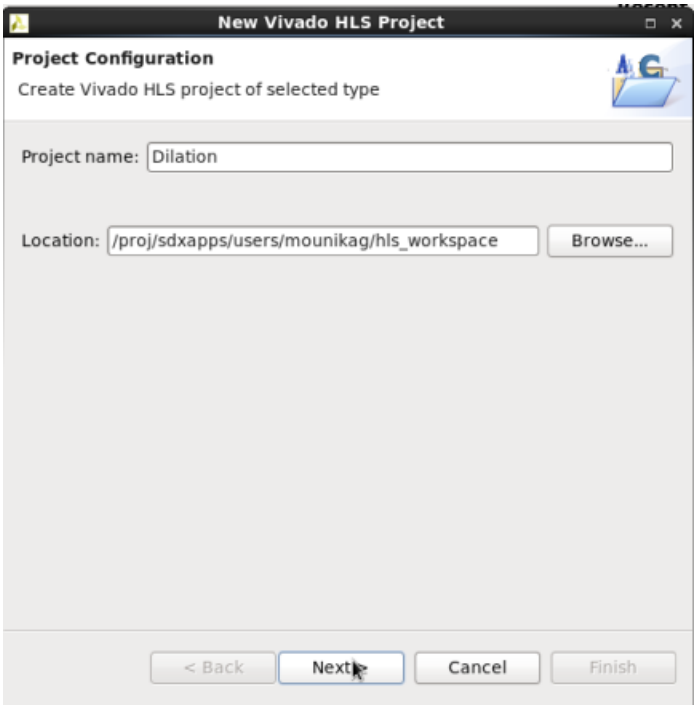
Using with GUI:

1. Open Vivado HLS in GUI mode and create a new project

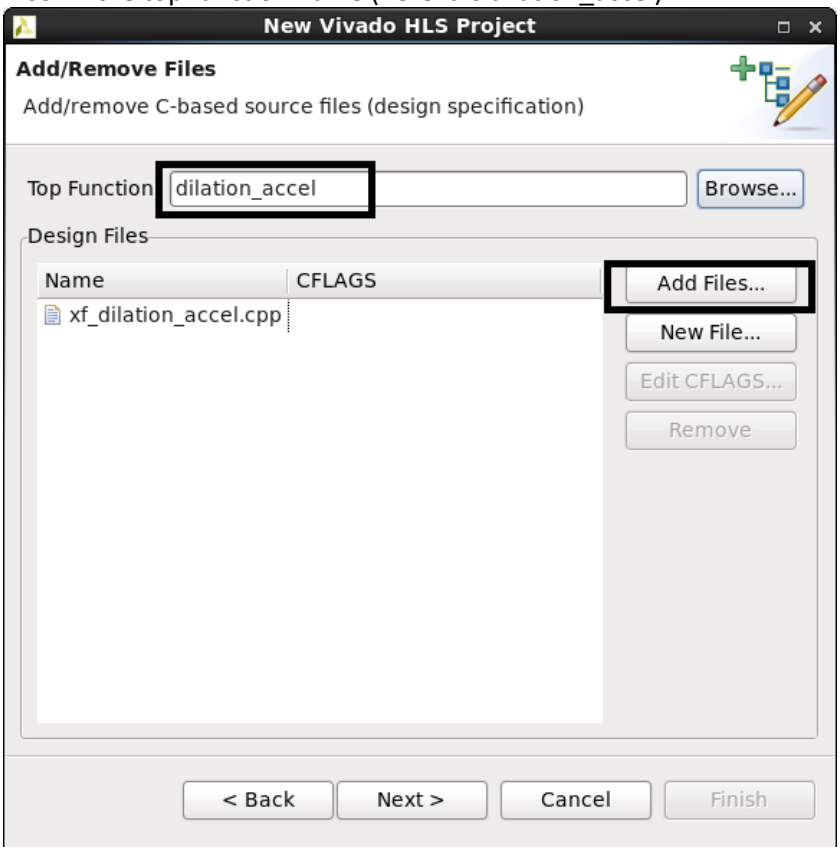


2. Specify the name of the project. For example **Dilation.**

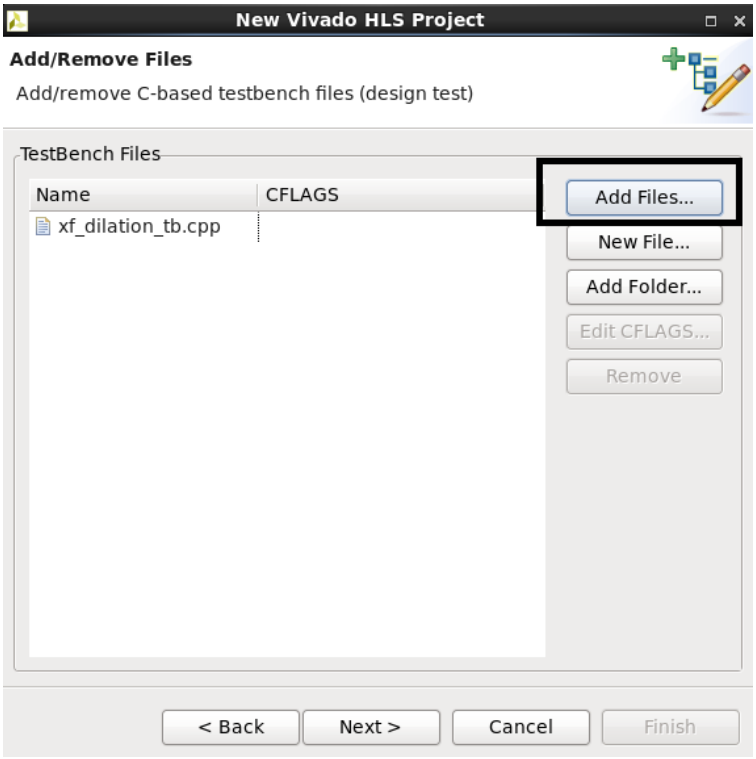
3. Click **Browse** to enter a workspace folder used to store your projects then click **Next.**



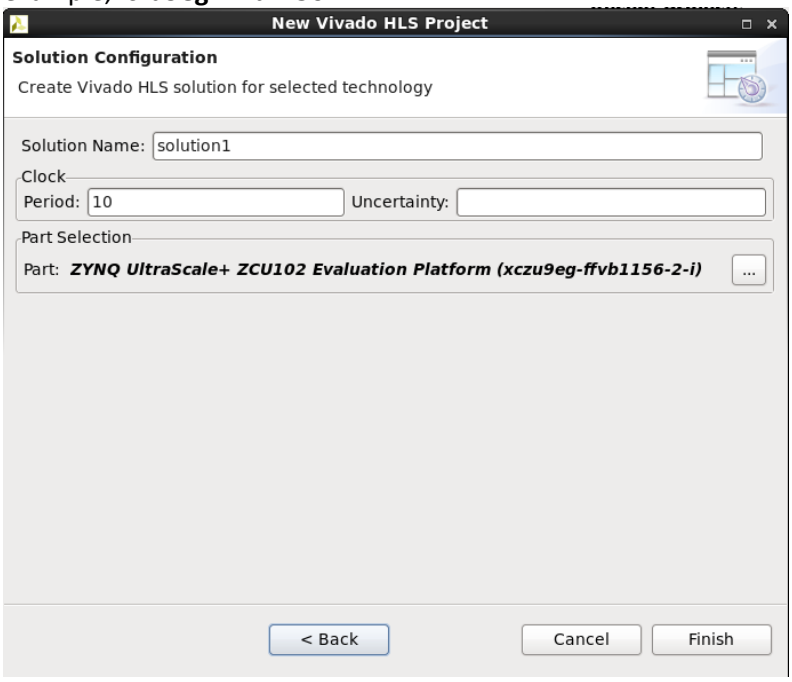
4. Under the source files section, add **accel.cpp** file which can be found in the **examples** folder. Also fill the top function name (here it is dilation_accel).



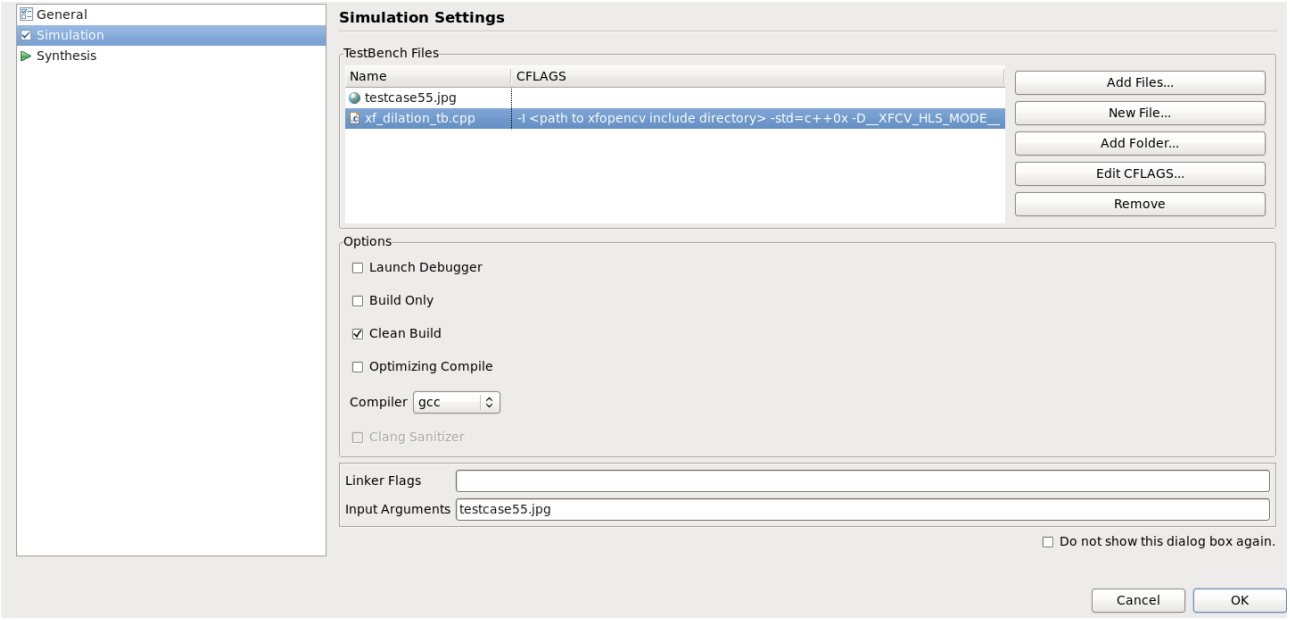
5. Click **Next.**
6. Under the test bench section add **tb.cpp**.



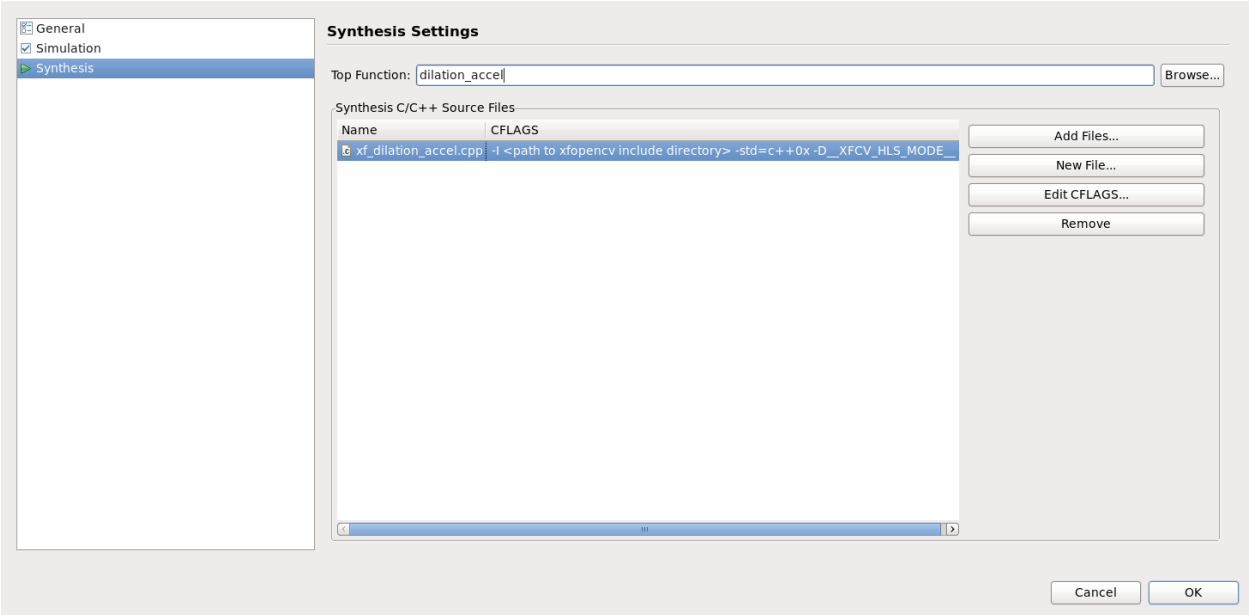
7. Click **Next.**
8. Select the **clock period** to the required value (10ns in example), select the suitable part. For example,**xczu9eg-ffvb1156-2-i.**



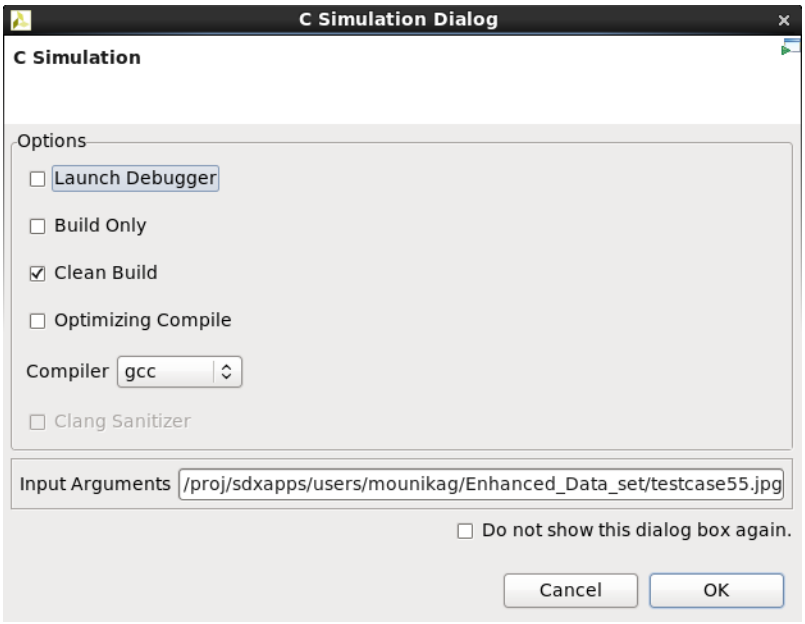
9. Click **Finish.**
10. Right click on the created project and select '**Project Settings'**
11. Select **Simulation** in the opened tab.
12. Files added under Test Bench section will be displayed. Select a file and click "**Edit CFLAGS**".
13. Enter "`-I<path-to-include-directory> -D__XFCV_HLS_MODE__ -std=c++0x`"



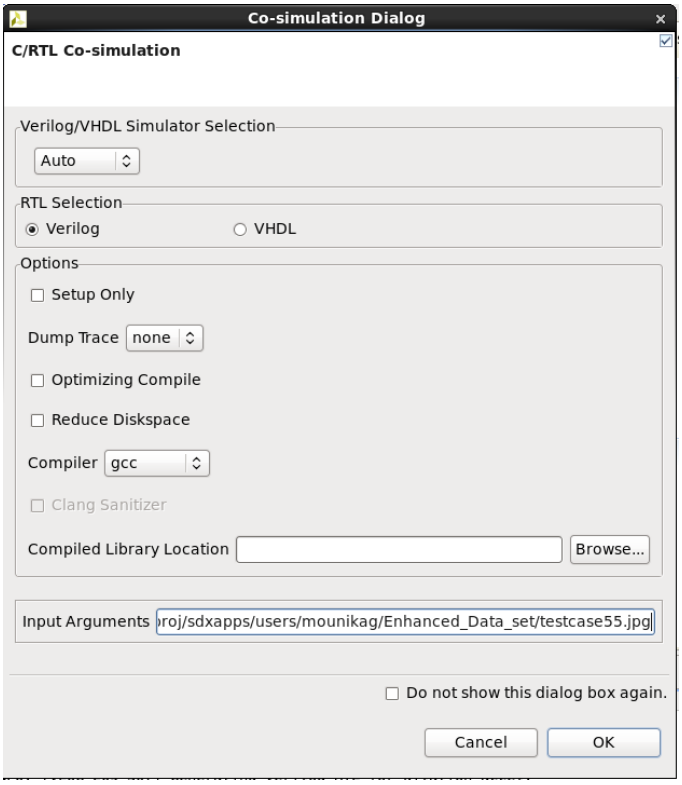
14. Click **OK.**
15. Repeat the above steps for all the files displayed.
16. Select Synthesis and follow the same process for all the files displayed.



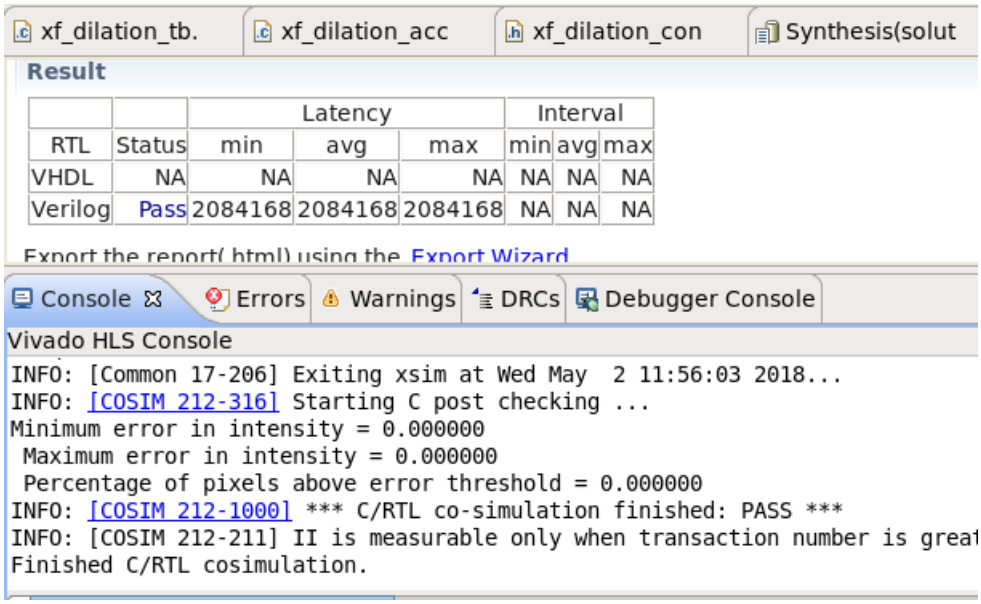
17. Run the C Simulation select **Clean Build** and specify the required **input arguments**.



18. Click **OK**
19. All the generated output files/images will be present in the `solution1->csim->build`
20. Run the **C Synthesis**.
21. Run the **Co-simulation** by specifying the proper input arguments.



22. The status of co-simulation can be observed on the console.



List of constraints for co-simulation:

There are few limitations in performing co-simulation of xfOpenCV functions.

1. Functions with multiple accelerators are not supported.
2. Compiler and simulator are the default ones in HLS. (gcc, xsim).
3. Memory mapped functions (warpaffine, perspective & edgelinking) do not support co-simulation.
4. Since HLS does not support multi-kernel integration, the current flow also does not support multi-kernel integration. Hence, the following functions and examples are not supported in this flow: pyramidal optical flow, Canny edge detection.

AXI Video Interface functions:

xfOpenCV has functions that will transform the xf::Mat into Xilinx Video Streaming interface and viceversa. xf::AXIvideo2xfMat() and xf::xfMat2AXIVideo() acts as video interfaces to the IPs of xfOpenCV functions in Vivado-IP integrator. cvMat2AXIvideoxf <NPC>, AXIvideo2cvMatxf<NPC> are used on the host side.

Functions and description:

Video Library Function	Description
AXIvideo2xfMat	Converts data from an AXI4 video stream representation to xf::Mat format.
xfMat2AXIvideo	Converts data stored as xf::Mat format to an AXI4 video stream.
cvMat2AXIvideoxf	Converts data stored as cv::Mat format to an AXI4 video stream
AXIvideo2cvMatxf	Converts data from an AXI4 video stream representation to cv::Mat format.

AXIvideo2xfMat

AXIvideo2xfMat function receives a sequence of images using the AXI4 Streaming Video and produces an xf::Mat representation.

API Syntax

```
template<int W,int T,int ROWS, int COLS,int NPC>

int AXIvideo2xfMat(hls::stream< ap_axiu<W,1,1,1> >&
AXI_video_strm, xf::Mat<T,ROWS, COLS, NPC>& img)
```

Template and Function Parameter Descriptions

Parameter	Description
W	Data width of AXI-stream. Recommended value is pixel depth.
T	Pixel type of the image. 1 channel (XF_8UC1). Data width of pixel must be no greater than **W**
ROWS	Maximum height of input image.
COLS	Maximum width of input image.
NPC	Number of Pixels to be processed per cycle; possible options are XF_NPPC1 and XF_NPPC8 for 1-pixel and 8-pixel operations respectively.

AXI_video_strm	HLS stream of ap_axiu(axi protocol) type.
img	Input image.

This function will return bit error of ERROR_IO_EOL_EARLY(1) or ERROR_IO_EOL_LATE(2) to indicate an unexpected line length, by detecting TLAST input.

For more information about AXI interface see (ug761)

xfMat2AXIvideo

The Mat2AXIvideo function receives an xf::Mat representation of a sequence of images and encodes it correctly using the AXI4 Streaming video protocol.

API Syntax

```
template<int W, int T, int ROWS, int COLS,int NPC>

int xfMat2AXIvideo(xf::Mat<T,ROWS, COLS,NPC>&
img,hls::stream<ap_axiu<W,1,1,1> >& AXI_video_strm)
```

Template and Function Parameter Descriptions

Parameter	Description
W	Data width of Axi-stream. Recommended value is pixel depth.
T	Pixel type of the image. 1 channel (XF_8UC1) is supported. The data width of pixel must be no greater than the **W**
ROWS	Maximum height of Output image.
COLS	Maximum width of Output image.
NPC	Number of Pixels to be processed per cycle; possible options are XF_NPPC1 and XF_NPPC8 for 1-pixel and 8-pixel operations respectively.
img	Output image
AXI_video_strm	HLS stream of ap_axiu(axi protocol) type.

This function returns the value 0.

Note: The NPC values across all the functions in a data flow must follow the same value. If there is mismatch it throws a compilation error in HLS.

cvMat2AXIvideoxf

The **cvMat2Axivideoxf** function receives image as cv::Mat representation and produces the AXI4 streaming video of image.

API Syntax

```
template<int NPC,int W>

void cvMat2AXIvideoxf(cv::Mat& cv_mat, hls::stream<ap_axiu<W,1,1,1> >&
AXI_video_strm)
```

Template and Function Parameter Descriptions

Parameter	Description
W	Data width of Axi-stream. Recommended value is pixel depth.
NPC	Number of Pixels to be packed and writing into stream. Possible options are XF_NPPC1 and XF_NPPC8 for 1-pixel and 8-pixel operations respectively.
AXI_video_strm	HLS stream of ap_axiu(axi protocol)type.
cv_mat	Input image

AXIvideo2cvMatxf

The Axivideo2cvMatxf function receives image as AXI4 streaming video and produces the cv::Mat representation of image

API Syntax

```
template<int NPC,int W>

void AXIvideo2cvMatxf(hls::stream<ap_axiu<W,1,1,1> >& AXI_video_strm,
cv::Mat& cv_mat)
```

Template and Function Parameter Descriptions

Parameter	Description
Parameter	**Description**

W	Data width of Axi-stream. Recommended value is pixel depth.
NPC	Number of Pixels to be packed and writing into stream. Possible options are XF_NPPC1 and XF_NPPC8 for 1-pixel and 8-pixel operations respectively.
AXI_video_strm	HLS stream of ap_axiu(axi protocol)type.
cv_mat	Output image

Usage Example with interface functions:

OpenCV-based testbenches use the cvMat2AXIvideoxf, AXIvideo2cvMatxf. These functions are commonly used to convert cv::Mat object data(input frame) into Axivideo stream.

The below code illustrates the usage of **dilation** on an image, using Axivideo interface functions.

```
xf_dilation_tb.cpp
int main(int argc, char** argv)
{
   cv::Mat out_img,ocv_ref, in_img,in_img1,diff;
   in_img = cv::imread(argv[1], 0);
   uint16_t height = in_img.rows; uint16_t width = in_img.cols;
   hls::stream< ap_axiu<8,1,1,1> > _src,_dst;
   cvMat2AXIvideoxf<NPC1>(in_img, _src);
   ip_accel_app(_src, _dst,height,width);
   AXIvideo2cvMatxf<NPC1>(_dst, in_img1);
   cv::imwrite("hls.jpg", in_img1);
   cv::imwrite("out_ocv.jpg", ocv_ref);
   return 0;
}
```

`cvMat2AXIvideoxf,AXIvideo2cvMatxf` functions fill frame or image data into AXI4 hls stream format. End user has to call the functions `AXIvideo2xfMat,xfMat2AXIvideo` to convert Axivideo streams to xf::Mat and vice versa.

User needs to create a wrapper which accepts the hls::streams as input/output. Dataflow mode is selected to enabling concurrent execution of the various processing functions, with pixels being streamed from one block to another. These `#pragma HLS INTERFACE`

directives expose the input and output streams as AXI4 Streaming interfaces.

xf_ip_accel.cpp

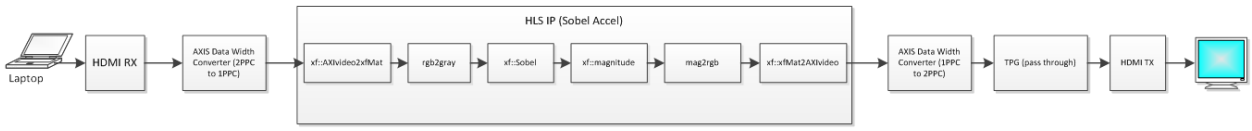
```
void dilation_accel(xf::Mat<TYPE, HEIGHT, WIDTH, NPC1>
&_src,xf::Mat<TYPE, HEIGHT, WIDTH, NPC1> &_dst);


void ip_accel_app (hls::stream< ap_axiu<8,1,1,1> >&
_src,hls::stream< ap_axiu<8,1,1,1> >& _dst,int height,int width)
{
#pragma HLS INTERFACE axis register both  port=_src
#pragma HLS INTERFACE axis register both  port=_dst

    xf::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgInput1(height,width);
    xf::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgOutput1(height,width);

#pragma HLS stream variable=imgInput1.data dim=1 depth=1
#pragma HLS stream variable=imgOutput1.data dim=1 depth=1
   #pragma HLS dataflow

   xf::AXIvideo2xfMat(_src, imgInput1);

   dilation_accel (imgInput1,imgOutput1);

   xf::xfMat2AXIvideo(imgOutput1, _dst);
}
```

The `AXIvideo2xfMat, xfMat2AXIvideo` are declared in **<path to include library/include/common/xf_infra.h>.**

The `cvMat2AXIvideoxf, AXIvideo2cvMatxf` are declared in **<path to include library/include/common/xf_axi.h>.**

Creating an image processing Pipeline application with AXI interface functions:

Pipeline applications can be created using the axivideo interface functions existing in the xfOpenCV library.



The sobel/magnitude pipeline application uses the laptop as HDMI source and displays the output on a monitor. The HDMI subsystem only supports 2 or 4 pixel per clock (PPC) whereas the AXIS2Mat/Mat2AXIS video IP HLS function only supports 1 PPC or 8PPC. To convert from 2 to 1pixel per clock another block AXIS data width converter needs to be added. The HDMI input gets filled into the axistream and this stream is provided as input to HLS IP(sobel_accel) containing the AXI interfaces. The pipeline is made up of the following functions:

1. AXIvideo2xfMat
2. rgb2gray
3. Sobel
4. Magnitude
5. mag2rgb
6. xfMat2AXIvideo

The following example defines the pipeline application

```
void sobel_accel(hls::stream< ap_axiu<24,1,1,1> >& _src,hls::stream<
ap_axiu<24,1,1,1> >& _dst, ap_uint<32> threshold)
{
#pragma HLS INTERFACE s_axilite port=threshold bundle=axi_lite
#pragma HLS INTERFACE s_axilite port=return bundle=axi_lite
#pragma HLS INTERFACE axis register both port=_src
#pragma HLS INTERFACE axis register both port=_dst
       xf::Mat<XF_8UC3, HEIGHT, WIDTH, NPC1> RGB_IN(HEIGHT,WIDTH);
#pragma HLS STREAM variable=RGB_IN depth=1 dim=1
       xf::Mat<XF_8UC1, HEIGHT, WIDTH, NPC1> Y(HEIGHT,WIDTH);
#pragma HLS STREAM variable=Y depth=1 dim=1
       xf::Mat<XF_16UC1, HEIGHT, WIDTH, NPC1> dstx(HEIGHT,WIDTH);
#pragma HLS STREAM variable=dstx depth=1 dim=1
       xf::Mat<XF_16UC1, HEIGHT, WIDTH, NPC1> dsty(HEIGHT,WIDTH);
```

```
#pragma HLS STREAM variable=dsty depth=1 dim=1
       xf::Mat<XF_16UC1, HEIGHT, WIDTH, NPC1> mag(HEIGHT,WIDTH);
#pragma HLS STREAM variable=mag depth=1 dim=1
       xf::Mat<XF_8UC3, HEIGHT, WIDTH, NPC1> RGB_OUT(HEIGHT,WIDTH);
#pragma HLS STREAM variable=RGB_OUT depth=1 dim=1



#pragma HLS DATAFLOW
      xf::AXIvideo2xfMat<24,XF_8UC3,HEIGHT,WIDTH,NPC1>(_src, RGB_IN);
      rgb2gray(RGB_IN,Y);
      xf::Sobel<XF_BORDER_CONSTANT,FILTER_WIDTH,XF_8UC1,XF_16UC1,HEIGHT,
WIDTH,NPC1>(Y, dstx,dsty);
      xf::magnitude<XF_L1NORM,XF_16UC1,XF_16UC1, HEIGHT, WIDTH,NPC1>(dstx, dsty,
mag);
      mag2rgb(mag,RGB_OUT,threshold);
      xf::xfMat2AXIvideo<24,XF_8UC3,HEIGHT,WIDTH,NPC1>(RGB_OUT, _dst);


}
```

Note: All the xfOpenCV library functions are verified against OpenCV-3.0.