

SDSoC, Computer Vision, and Machine Learning

“Don’t Panic!”

Hitchhiker’s Guide to the Galaxy
DOUGLAS ADAMS

Rob Armstrong
Sr. Product Marketing Manager, Xilinx, Inc.
robert.armstrong-jr@xilinx.com



Contents

	XDF 2018 SDSoC & Machine Learning Lab	1
1	Introduction	1
2	Connecting to Amazon AWS	2
	2.1 Log in to AWS and Start the Instance	2
	2.2 Connect to the AWS AMI Using RDP	4
3	Provided Design Files	6
4	Design Details	6
	4.1 Theory of Operation: Pixel Format Conversion	7
5	Defining the Software Architecture	8
6	Building the SDSoC Design	10
7	Building the GStreamer Plugins	15
	7.1 Face Detector	15
	7.2 Pedestrian Detector	16
	7.3 UYVY to RGB	16
8	Assembling the Boot Files	18
9	Running on the Board	19
10	Summary	21

XDF 2018: SDSoC, Machine Learning, and Computer Vision

1 Introduction

Welcome to the XDF 2018 SDSoC, machine learning, and computer vision workshop!

The Xilinx SD x development environment is uniquely suited to developing systems combining hardware-accelerated machine learning, sensor input from a variety of sources, and accelerating custom algorithms using FPGA fabric. By leveraging the tight coupling between the CPUs and FPGA fabric in Xilinx's ACAP and SoC devices, designers have the flexibility to adapt their hardware and platforms to achieve unparalleled acceleration.

In this lab we'll take video input from a USB3 webcam, run it through the Deep-learning Processing Unit (DPU), and detect faces in the input video stream using the DenseBox neural network as described in figure 1. We will then output the video with an overlay onto a DisplayPort-connected monitor. All of this will be built targeting the low-cost, high performance, and low-power Ultra96 board from Avnet on a Xilinx Zynq UltraScale+ ZU3 device.



Figure 1: Desired System Top-Level Connectivity

To move video in the system we will use the popular open-source *gstreamer* framework to sequence processing operations, all running on Linux on the quad-core ARM Cortex-A53 processing subsystem.

Note that no prior experience with SDSoC is necessary for this lab - all instructions needed are provided, although we do assume a certain degree of background knowledge on C++ programming. If you are unsure how to approach a particular step or sequence of operations please reach out to one of the lab assistants standing by to help. Alternatively, advanced users who are familiar with the SDSoC tool flow and environment may wish to experiment on their own.

2 Connecting to Amazon AWS

In this lab exercise you will start and connect to an AWS EC2 m5.8xlarge instance and connect to it using either SSH or the Remote Desktop client. Once connected you will be able to run the Xilinx tools with all needed scripts and configurations.

For this event, each registered participant will be required to start their own AWS instance.

2.1 Log in to AWS and Start the Instance

You should have received a piece of paper which has the Account ID, IAM user name, and password for a unique AWS AMI. If you don't have it, or have lost it, please raise your hand and ask one of the lab assistants for help.

Begin by using your web browser to visit this link:

<https://console.aws.amazon.com/ec2/v2/home?region=us-west-2#Instances:tag:Name=your IAM user name;sort=tag:Name>

For example, if your user name is "user7", the correct URL would be:

<https://console.aws.amazon.com/ec2/v2/home?region=us-west-2#Instances:tag:Name=user7;sort=tag:Name>

Log in with your assigned account ID, user name, and password as shown in figure 2.

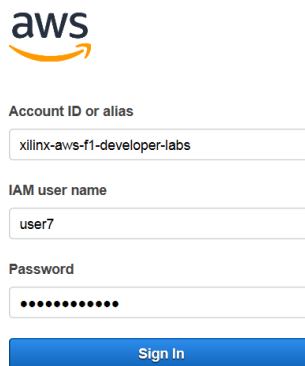
The image shows the AWS login interface. At the top is the AWS logo. Below it are three input fields: 'Account ID or alias' with the value 'xilinx-aws-f1-developer-labs', 'IAM user name' with the value 'user7', and 'Password' with masked characters. A blue 'Sign In' button is at the bottom.

Figure 2: AWS Login Screen

You will see a list of AWS instances. Select the instance associated with your user name, noting that there are many attendees registered for XDF and you may have to scoll a bit to find yours (there is a search/filter function available at the top of the screen). One you've located it click **Actions** → **Instance State** → **Start** as in figure 3.

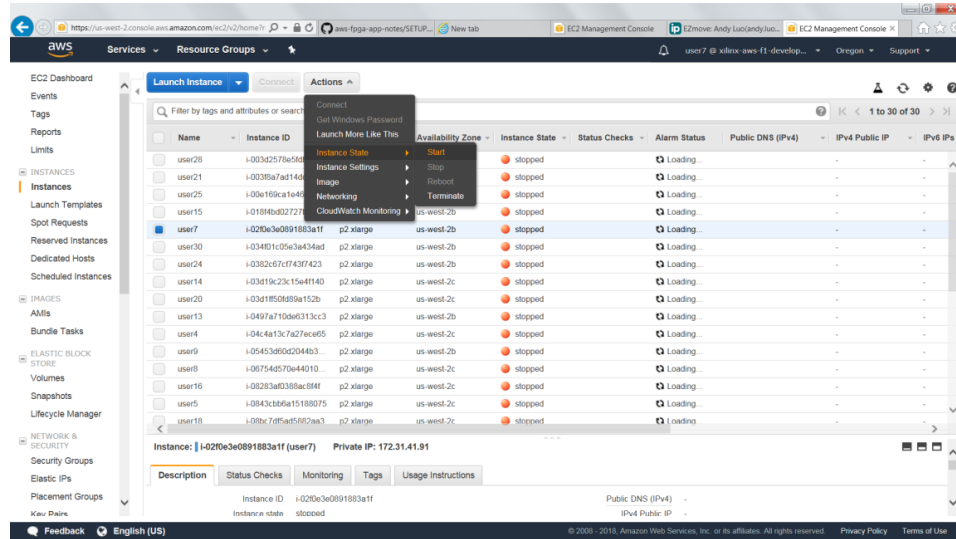


Figure 3: AWS Instance Start

Each instance takes approximately 10 to 20 seconds to start, and you will need to refresh your browser in order to see the status update. Once the instance has booted the state will display as “running” and you will see an IPv4 public IP address associated with it as in figure 4. Take note of this address as we will use it in subsequent steps to connect to the instance and access the lab software environment.

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
user28	i-003d2578e5db839d	p2.xlarge	us-west-2c	stopped		Loading...		-
user21	i-0038a7ad14dc6c8b	p2.xlarge	us-west-2c	stopped		Loading...		-
user25	i-00e169ca1e46ab1c9	p2.xlarge	us-west-2c	stopped		Loading...		-
user15	i-018f4bd02727b0829	p2.xlarge	us-west-2b	stopped		Loading...		-
user7	i-02f0e3e0891883a1f	p2.xlarge	us-west-2b	running	2/2 checks...	Loading...	ec2-34-211-165-84 us-...	34.211.165.84
user30	i-034f01c05e3a434ad	p2.xlarge	us-west-2b	stopped		Loading...		-
user24	i-0382c67cf743f7423	p2.xlarge	us-west-2b	stopped		Loading...		-
user14	i-03d19c23c15e4f140	p2.xlarge	us-west-2c	stopped		Loading...		-

Figure 4: AWS Instance Running with IP Address

There are two ways to connect to the instance: SSH and RDP. This lab requires the use of the GUI and Xilinx has found RDP connectivity to generally be more responsive than SSH tunneling and/or VNC for AWS instances and RDP clients are available on most systems, so we will provide instructions for RDP only here.

2.2 Connect to the AWS AMI Using RDP

The instance you started should be configured with an RDP server. To connect:

Step 1: From your local laptop, start a remote desktop client

- **Windows:** press the Windows key and type “remote desktop”
 - You should see the “Remote Desktop Connection” application in the list of programs
 - If you do not, alternatively you can type **mstsc.exe** in the Windows run prompt
- **Linux:** RDP clients such as Remmina and Vinagre are suitable
- **macOS:** Use the Microsoft Remote Desktop app from the Mac App Store

Step 2: In the remote desktop client, enter the public IPv4 address of your instance. If you are unsure where to find it please refer back to figure 4.

Step 3: **IMPORTANT:** Before connecting, set your remote desktop client to use **24-bit or lower color depth**. This is a shared network environment with many people on WiFi, your performance (and theirs) will be degraded with higher bandwidth utilization.

On Windows, in the bottom-left corner of the connection prompt click **Options**, then select the Display tab and set Colors to True Colors (24-bit).

Step 4: Click **Connect** (or equivalent) to connect to the remote system.

Step 5: **Note:** You may see a message about untrusted connection certificates. If so, click **Yes** to dismiss this message. Your remote desktop client should prompt you to log in.

Step 6: Log in with the following credentials:

- User: **ubuntu**
- Password: **xdf_sdsoc**

Note that it is possible that the username and password you were provided for your use during the session may not match the above. In the event of a discrepancy you must use the **provided login and password**.

It is possible that the RDP connection may fail on the first attempt. Typically connecting a second time resolves the issue.

Step 7: Open a new terminal as in figure 5.

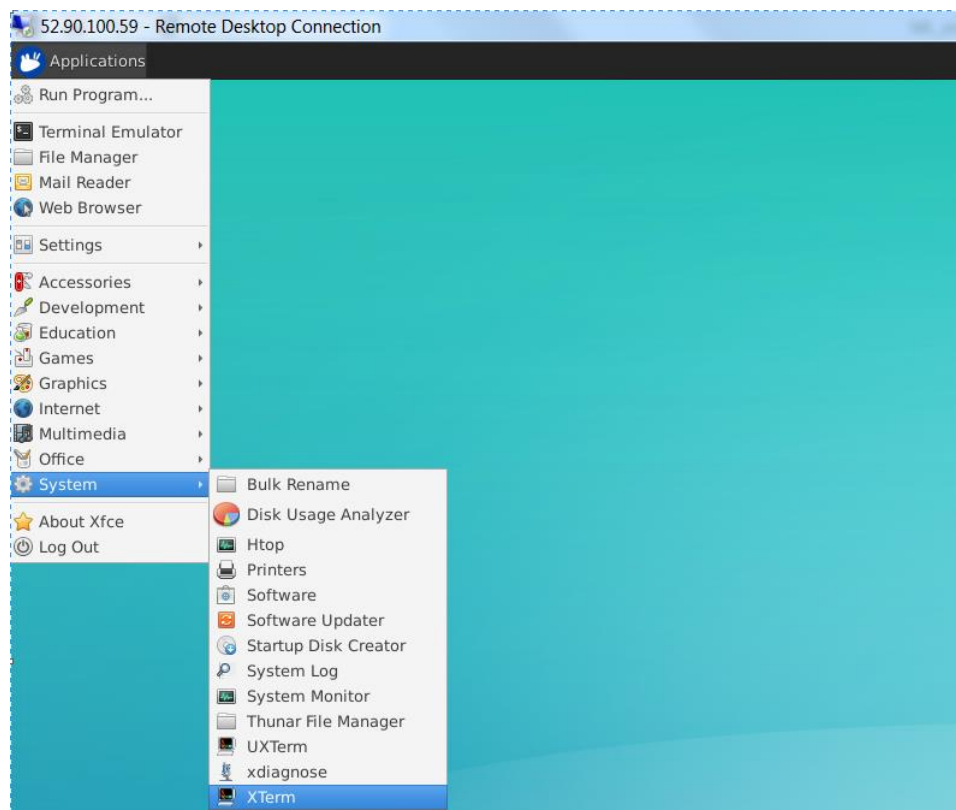


Figure 5: Opening a New Terminal over RDP

3 Provided Design Files

All design files for this lab exist under the the `~/XDF_Labs/SDSoC/` directory on the AWS AMI you were assigned.

Under this top-level source directory you will find four primary directories: **lab_files**, **platform**, **workspace**, and **solution**. The **lab_files** directory contains the source files you will import to begin working on the lab. The **platform** directory contains all of the files necessary to target the Ultra96 board with SDSoC, including the contents of the root file system needed for cross-compiling the gstreamer plugins. The **workspace** directory is currently empty but will be used for our target workspace, and the **solution** directory contains, as its name implies, a ready-made solution to the lab that is complete and working. It can be used for reference if you get stuck on any step.

4 Design Details

This lab targets an effectively empty hardware design containing only the clocking and reset infrastructure needed to enable the device. For our purposes here we will effectively ignore it as it consumes nearly no FPGA resources. The USB3 and DisplayPort interfaces are implemented using the hardened IP in the Zynq UltraScale+ Processing system. We will interact with them using standard drivers built into the Linux kernel.

However, engineering unfortunately seldom lends itself to simple block diagrams. In this example we have the following constraints:

- Our neural network has been trained on 640x480 images using the RGB pixel format.
- Our USB3 webcam will produce raw data, and at this resolution, but in the YUV 4:2:2 pixel format.
- The DisplayPort controller in the Zynq UltraScale+ PS block requires that the output frame buffers be aligned to a 256-byte stride.

From these constraints we can draw two conclusions: We must convert between the YUV 4:2:2 pixel format and RGB, and we must put the output format into something that can be displayed by the hardware. To that end we will work on the “real” block diagram shown in figure 6.

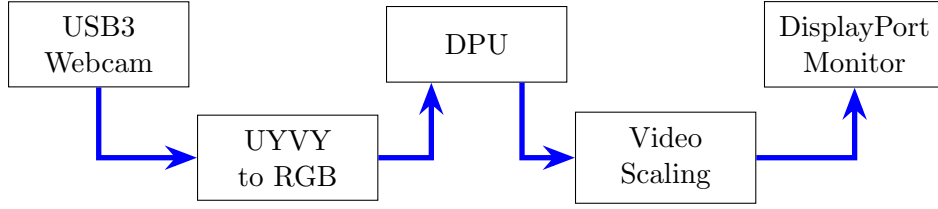


Figure 6: Full System-Level Connectivity

In this example design we will implement the three blocks not composed of hard silicon (UYVY to RGB conversion, the DPU, and video scaling) with a mix of accelerated hardware functions and software functions using the SDSoC tool.

Before beginning to build the design with SDSoC it's important to think about which blocks you intend to implement in hardware vs. which blocks you intend to implement in software. In a real system you would likely want to implement all of these as hardware blocks for performance, but perhaps you have other algorithms to accelerate, etc. For this lab we will use:

- The DPU IP from Xilinx, which is provided as a “C-Callable IP Library” for the SDSoC tool. Xilinx provides a standard API to interact with the library, and the relevant portions of the hardware functionality are automatically accelerated using the FPGA fabric.
- To convert from YUV 4:2:2 (hereinafter referred to as UYVY) to RGB we will use a standard C++ function accelerated into the FPGA fabric.
- To scale the video to the display we will use software scaling in gstreamer. This is **not** the way to implement the system for maximum performance; our goal is to showcase the interoperability of SDSoC-accelerated components along with software processing. For best results please refer to the Xilinx xf::OpenCV hardware-accelerated OpenCV library.

4.1 Theory of Operation: Pixel Format Conversion

Our See3Cam camera from E-Con Systems will provide UYVY data at up to 1080p resolutions, but to avoid an input scaling step we will configure it to provide raw UYVY data at our requested 640x480 resolution. We then need to convert that data in memory to RGB data that our neural network has been trained to understand.

Fortunately, converting between UYVY and RGB is a simple operation that works very nicely with the stream-based processing capabilities of an FPGA. In memory the UYVY data is laid out as in figure 7 with each value represented by a single byte.

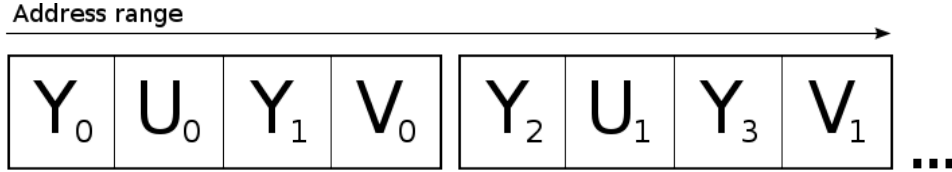


Figure 7: UYVY Memory Layout

Because UYVY is a sparse format we can't convert pixel-by-pixel but rather using a full four-byte word (or, in other words, two pixels to two pixels). The equations for the conversion are as follows:

$$R_0 = 1.164 \times (Y_0 - 16) + 2.018 \times (U_0 - 128)$$

$$G_0 = 1.164 \times (Y_0 - 16) - 0.813 \times (V_0 - 128) - 0.391 \times (U_0 - 128)$$

$$B_0 = 1.164 \times (Y_0 - 16) + 1.596 \times (V_0 - 128)$$

$$R_1 = 1.164 \times (Y_1 - 16) + 2.018 \times (U_0 - 128)$$

$$G_1 = 1.164 \times (Y_1 - 16) - 0.813 \times (V_0 - 128) - 0.391 \times (U_0 - 128)$$

$$B_1 = 1.164 \times (Y_1 - 16) + 1.596 \times (V_0 - 128)$$

This pattern repeats throughout the image, making implementation with a **for** loop simple.

Using fixed-point representations of the constant values in the FPGA fabric results in an extremely small and efficient implementation that's able to produce a new pixel value on each clock cycle, converting the values easily without consuming valuable CPU processing time.

5 Defining the Software Architecture

Now that you have a general understanding of the system we will address the “nuts and bolts” of getting the system constructed. Video systems built with **gstreamer** are built using **pipelines**, which are strings of processing elements, each one of which performs a particular function. Many functions and interfaces are available, but the details of **gstreamer** are beyond the

scope of this lab. Suffice it to say that we will be building a pipeline with the topology indicated in figure 8.

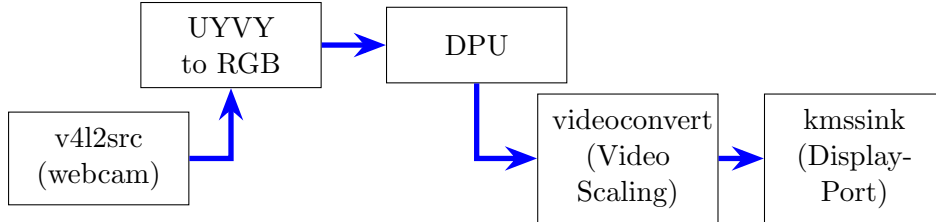


Figure 8: GStreamer Pipeline

Each block in figure 8 will be implemented with a pre-compiled *shared object file*, or .so file, in Linux that defines the interfaces, block behavior, etc. There are many topologies we can implement with SDSoC, but we will implement something like the following where both the UYVY to RGB and DPU functions call into a single hardware-accelerated library. This isolates, to the degree possible, the hardware implementation from the gstreamer implementation. Our full topology will look like figure 9.

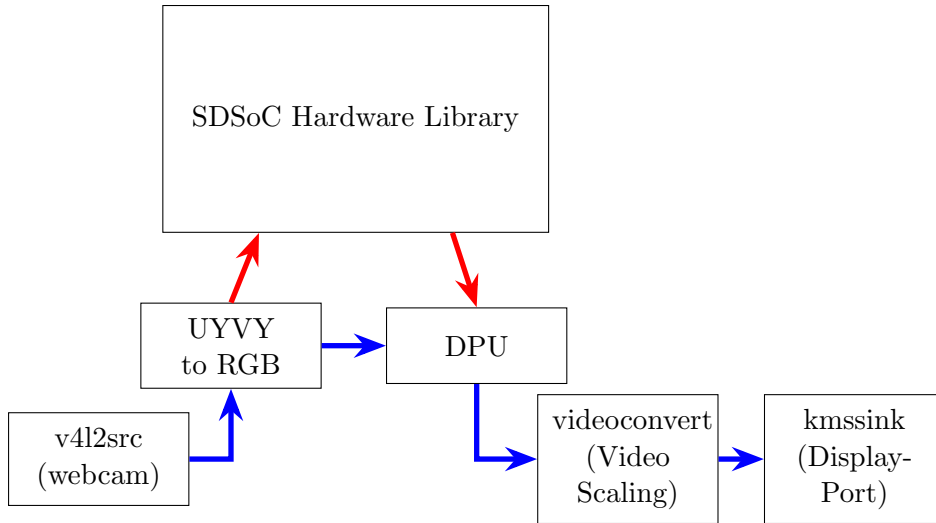


Figure 9: GStreamer Libraries

Now that we have a general understanding of the IPs we want to create and their interfaces, it's time to package them up in a meaningful way for

use in SDSoC. We will create a single library, **libdpucore130_1152.so**, that contains both the DPU C-Callable IP library functions and the hardware-accelerated pixel format conversion function. This library will consume the **libdpu.a** library which contains the DPU IP (and this is the general SDSoC flow for embedded hardware libraries). If you are unfamiliar, .a file in C or C++ is a *static archive*. This is a pre-compiled archive that contains, in our case, both pre-compiled C++ function stubs and the source files to recreate the DPU IP. This means that when using C-Callable IP with SDSoC you don't need to worry about copying around the original IP sources or dealing with instantiating the IP directly into the Vivado development environment.

Xilinx provides pre-defined C++ functions that will map onto the top-level interfaces of the DPU IP. For our lab we will add in additional functions to enable the additional processing flows.

6 Building the SDSoC Design

Begin by connecting to the AMI containing the lab source files. The software environment has already been pre-configured, however, we need to set an environment variable to point to the cross-compilation target file system. This is to allow the development environment to parse and link against the headers, libraries, etc. that will be present on our target. This **sysroot** has been pre-built with the Xilinx PetaLinux toolchain.

From a command prompt, enter the following (note that for those unfamiliar with Linux syntax, the characters shown in the “export SYSROOT...” line are the backtick, typically found next to the number one on US keyboards, and not the single quote):

```
cd ~/XDF_Labs/SDSoC/platform/aarch64-xilinx-linux
export SYSROOT=`pwd`
```

You will want to double check that the \$SYSROOT environment variable is set correctly as any errors here will result in compilation errors later. Check with the “echo” command to ensure that it's correct:

```
echo $SYSROOT
/home/ubuntu/XDF_Labs/SDSoC/platform/aarch64-xilinx-linux
```

Then, launch the SDx development environment by running the command ‘sdx’ from the command line. When prompted, as in figure 10, select

the `/home/ubuntu/XDF_Labs/SDSoC/workspace` directory for your workspace. Note that you can also launch the tool from the command line and directly point to a workspace by running ‘`sdx -workspace <path to workspace>`’, so in our case you can launch the tool by running:

```
sdx -workspace ~/XDF_Labs/SDSoC/workspace
```

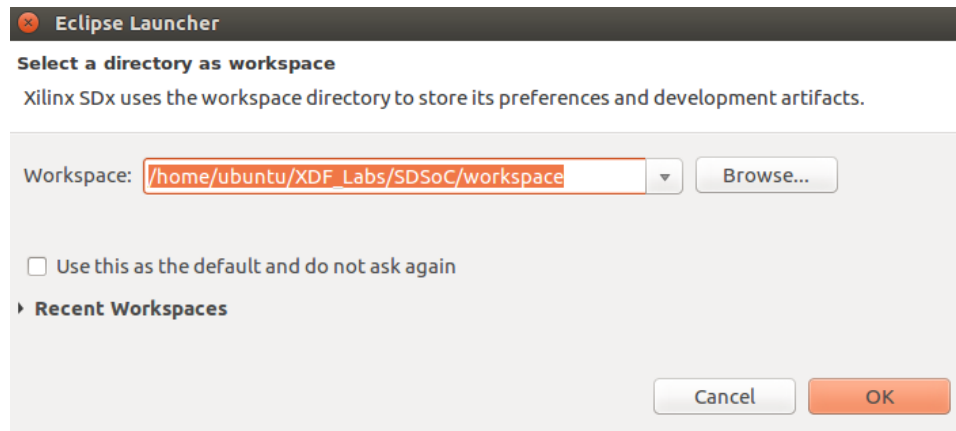


Figure 10: SDx Workspace Selection Dialog

Starting points for all applications have been provided. From the SDx welcome screen, select the **Import Project** link. In the subsequent dialog select “Eclipse workspace or zip file” as in figure 11. If you have accidentally closed the welcome screen please continue to the next paragraph, otherwise skip to the paragraph immediately following figure 12.

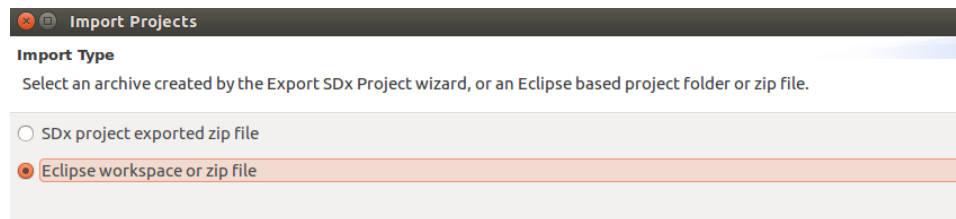


Figure 11: SDx Import Projects Dialog from the Welcome Screen

If you have closed the initial welcome screen, you will be presented with a blank workspace. Right-click within the “Project Explorer” pane and

select “Import”. To launch the import wizard, expand the “General” tab and select the “Existing Projects into Workspace” option as in figure 12.

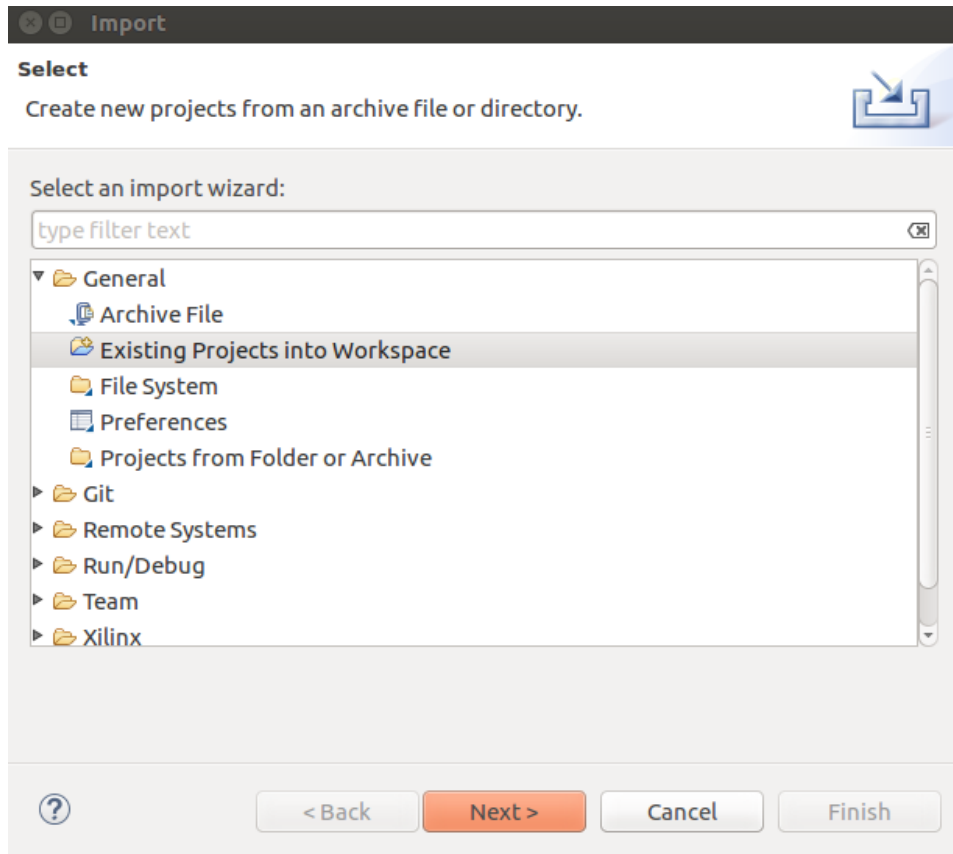


Figure 12: SDx Import Projects Dialog

From either pathway, click “Next”, then with the “Select root directory” radio button selected browse to the directory

/home/ubuntu/XDF_Labs/SDSoC/lab_files.

Ensure that the following projects are selected:

- ☐ dpucore130_1152
- ☐ gstdxfacedetect

- ☐ gstsdxpeditriandetect
- ☐ gstsdxyvy2rgb

NOTE: Also ensure that the “Copy projects into workspace” option is selected.

Your dialog should look similar to that shown in figure 13.

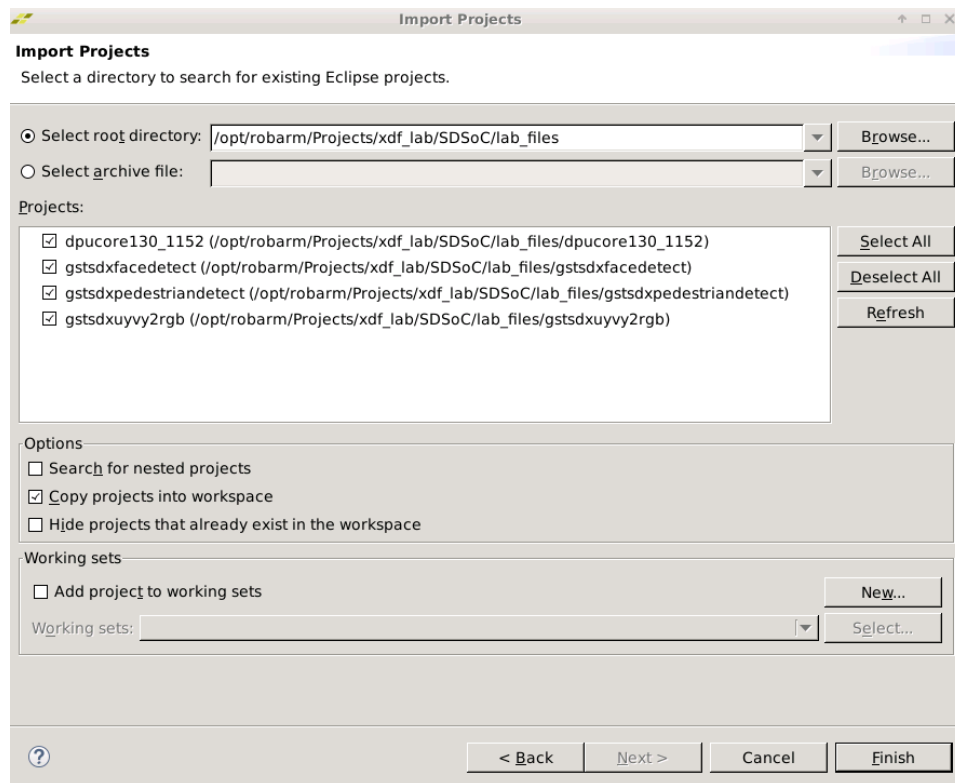


Figure 13: SDx Import Projects Dialog (Completed)

Click “Finish” to import all of the projects. Please note that two directories have been pre-installed into the workspace that do not exist by default. The first, a **libs** directory, contains support libraries for gstreamer and the DNNDK that we will link against to build our application. The second, an **ipcache** directory, contains cached hardware synthesis results to help speed up our implementation run (the AMI we’re using for this lab has a relatively small number of processor cores, and our lab time is short).

In the SDx GUI, expand the **dpucore130_1152** project (if it isn't already) and double click on the **project.sdx** file to open it.

The dpucore library includes the DPU and its associated software by default, but we now need to add our UYVY to RGB function. Right click on the 'src' folder and select "Import", then "General" → "File System". Browse to the directory

/home/ubuntu/XDF_Labs/SDSoC/lab_files/template

and select the file **uyvy2rgb.cpp**, importing it into the **src** folder of the DPU library.

Open the file by double clicking on it. Note that it contains the function to accelerate our UYVY to RGB conversion, called *uyvy2rgb_accel*. However, also note the TODO notice at the bottom. Every accelerated function in SDSoC must have one (or more) **call sites**. In other words, it can't be accelerated in a vacuum, it must be called by something. Add a wrapper around that function (*hint: this is just the same function with a different name, passing its data directly to the uyvy2rgb_accel function*). Something like listing 1.

```
// Caller for the conversion function
void uyvy2rgb(uyvy_data *uyvy,
              rgb_data *rgb,
              int height,
              int width)
{
    uyvy2rgb_accel(uyvy, rgb, height, width);
}
```

Listing 1: Call Site for uyvy2rgb_accel

Save the file. Our last step is to tell SDx we want to move the *uyvy2rgb_accel* function to hardware. To do that, from the main project page (which can be opened by double clicking on **project.sdx**) click on the lightning bolt icon (circled in figure 14 to select hardware functions to accelerate, and choose 'uyvy2rgb_accel' from the list. Please note that when the dialog box opens the source cache is refreshed; EBS network latency can make this process take roughly 30-45 seconds so please wait a moment if the dialog box comes up with an empty list. Also ensure that both the "Data motion network

clock frequency” and “Clock Frequency (MHz)” settings for the accelerator are both 300 MHz, as in figure 14.

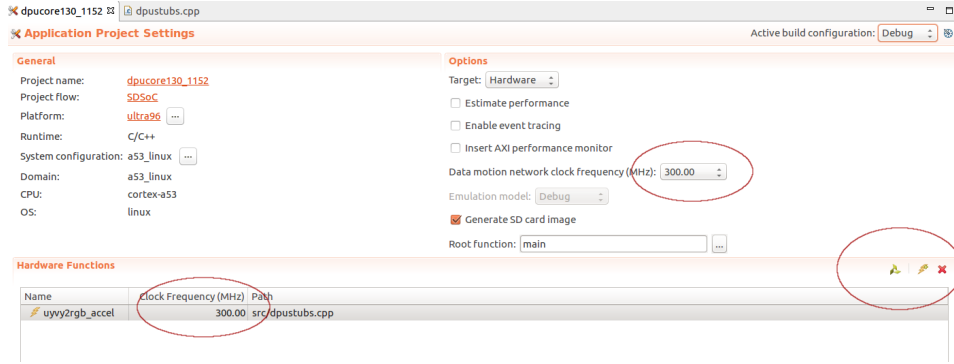


Figure 14: SDx Accelerator Configuration

Click the hammer icon (build) at the top to start the build process. This will take some time to complete. When it’s done we will have a file **libdpucore130_1152.so** that contains the software API to communicate with the DPU, along with our hardware-accelerated color conversion function. We can now link the other libraries against it.

NOTE: AWS resources are shared with other AWS users in the data center, including elastic block storage (EBS). Processes running over EBS may become I/O bound and take longer than they would on a local desktop machine. You may experience longer than normal build times when running this lab in the cloud. To save lab time, please cancel the build at this point and switch your workspace to the provided workspace with pre-built hardware by selecting **File** → **Switch Workspace** → **Other** and browsing to it. The workspace containing the pre-built hardware function is here:

~/XDF_Lab/SDSoC/workspace_prebuilt

7 Building the GStreamer Plugins

7.1 Face Detector

Double click on the **gstsdxfacedetect** project in the “Project Explorer” pane to open it, then click the build icon at the top of the screen. This project should build in seconds.

7.2 Pedestrian Detector

Double click on the **gstsdxpeditriandetect** project in the “Project Explorer” pane to open it, then click the build icon at the top of the screen. This project should build in seconds.

7.3 UYVY to RGB

Double click on the **gstsdxuyvy2rgb** project in the “Project Explorer” pane to open it. This project will build similarly quickly, but we have to make a quick modification first. Most of a gstreamer plugin is boilerplate; code that exists to interoperate with the library but that doesn’t impact the overall functionality (of course this is a gross simplification, but for our purposes in this lab it’s sufficiently accurate). The important part is actually calling the hardware-accelerated function!

The project is already set up to link against the .so file we generated earlier, but we don’t call the function in the C code. Under the ‘src’ directory of the project open the **gstsdxuyvy2rgb.cpp** file and browse to the TODO line (line 121). Here, call your function you defined in the hardware library. It should look something like the function on line 28 of listing 2.

```

1  static GstFlowReturn
2  gst_sdx_uyvy2rgb_process_frames(GstSdxBase    * base,
3                                  GstSdxFrame ** in_frames,
4                                  GstSdxFrame  * out_frame)
5  {
6      GstVideoInfo  *info = NULL;
7      GstSdxFrame   *in_frame;
8      GstVideoFrame *in_vframe, *out_vframe;
9      uyvy_data     *in_data;
10     rgb_data       *out_data;
11
12     g_return_val_if_fail(in_frames != NULL &&
13                           out_frame != NULL, GST_FLOW_ERROR);
14
15     in_frame = in_frames[0];
16     if (in_frame == NULL) {
17         GST_WARNING_OBJECT (base, "uyvy2rgb input frame is invalid");
18         return GST_FLOW_ERROR;
19     }
20
21     in_vframe = &in_frame->vframe;
22     out_vframe = &out_frame->vframe;
23     info       = &out_frame->info;
24     in_data    = (uyvy_data *) GST_VIDEO_FRAME_PLANE_DATA (in_vframe, 0);
25     out_data   = (rgb_data *) GST_VIDEO_FRAME_PLANE_DATA (out_vframe, 0);
26
27     // TODO: Add a function call to the UYVY accelerator here
28     uyvy2rgb(in_data, out_data, GST_VIDEO_INFO_HEIGHT (info),
29             GST_VIDEO_INFO_WIDTH (info));
30     /******
31
32     GST_DEBUG_OBJECT(base, "uyvy2rgb input frame processed");
33
34     return GST_FLOW_OK;
35 }

```

Listing 2: GStreamer “Process Frames” function calling our accelerator

With this function called, click the build icon at the top once again. This library should also build in a matter of seconds.

8 Assembling the Boot Files

We have now build a collection of .so files and a bitstream to run on the board, but these need to be collected into an SD card image for boot. We can collect them from each project build area into a directory to copy to an SD card image, as follows. If you are using a different workspace than workspace_prebuilt then please ensure you use the correct path for your initial ‘cd’ command.

```
cd ~/XDF_Labs/SDSoC/workspace_prebuilt
cp -r dpucore130_1152/Debug/sd_card .
cp libs/* sd_card/
cp gstdsxfacedetect/Debug/libgstdsxfacedetect.so sd_card/
cp gstdsxpeditriandetect/Debug/libgstdsxpeditriandetect.so sd_card/
cp gstdsxuyvy2rgb/Debug/libgstdsxuyvy2rgb.so sd_card/
```

We also have an optional shell script available that can be run on the board to automate the subsequent section. If you’re interested in the board setup that must be done to install and run custom gstreamer plugins then please ignore it and proceed on, but if you want to add it to your SD card you can install it thusly:

```
cp ../solution/sd_card/facedetect.sh sd_card/
cp ../solution/sd_card/pedestrian.sh sd_card/
```

Copy the complete contents of the **sd_card** directory onto your SD card with scp, sftp, or a similar SSH-based file transfer utility. There are many available and it’s beyond the scope of this document to provide instructions on each, but FileZilla on Windows and Linux (or command line tools on macOS/Linux) are reliable options. If you don’t have an SD card slot available, or are otherwise unable to perform this step, please ask one of the lab assistants for help (we can either help you transfer the files or provide a pre-completed SD card image).

9 Running on the Board

Connect the serial terminal of the Ultra96 board to your laptop. Your OS may need to automatically download the drivers for the USB UART device. Open a terminal to the tty or COM port created by the UART driver at 115200 kbps. Insert the SD card into the Ultra96 board and power it on. You should see boot messages displayed on the terminal. Do not interrupt the boot process.

Once the board boots, log in with the username **root** and password **root**.

Before launching our gstreamer pipeline we need to set the display to our desired resolution. By default the Linux kernel will auto-negotiate the preferred resolution of our display, which in this case is likely to be 1080p. For our design we want to output 720p. To force the resolution change enter the following command on the command line. This may seem cryptic, but `modetest` is a standard Linux application commonly used to force displays to different resolutions or perform other similar DRM/KMS setup tasks:

```
modetest -M xlnx -s 38@36:1280x720-60@RG24 -w 35:alpha:1 &
```

At this point if you are only interested in seeing the application run please run the **facedetect.sh** script from the command prompt:

```
source /media/card/facedetect.sh
```

To run the pedestrian detector, kill it with CTRL-C (Ĉ) and launch:

```
source /media/card/pedestrian.sh
```

Otherwise, if you are interested in manually performing the steps please continue on. To suppress kernel printk messages, which are output by default by the SDx runtime and which will clutter the terminal, run:

```
dmesg -n 1
```

We also need to install the shared objects to the correct places in the file system. Run the following commands:

```

cp libd* /usr/lib
cp libgstsdxallocator.so /usr/lib
cp libgstsdxbase.so /usr/lib
cp libgstsdxyvy2rgb.so \
  libgstsdxfacedetect.so \
  /usr/lib/gstreamer-1.0

```

Finally, we can launch the gstreamer pipeline for face detection:

```

gst-launch-1.0 \
  v4l2src io-mode=4 ! \
  video/x-raw,format=UYVY,width=640,height=480,framerate=30/1 ! \
  sdxyvy2rgb ! \
  rawvideoparse format="bgr" width=640 height=480 framerate=30 ! \
  sdx facedetect ! \
  videoscale ! \
  video/x-raw,format=BGR,width=1280,height=720 ! \
  kmssink driver-name=xlrx sync=false

```

You should see output from the webcam piped to the display, with faces clearly outlined by bounding boxes.

We can now launch the pedestrian detection similarly:

```

gst-launch-1.0 \
  v4l2src io-mode=4 ! \
  video/x-raw,format=UYVY,width=640,height=480,framerate=30/1 ! \
  sdxyvy2rgb ! \
  rawvideoparse format="bgr" width=640 height=480 framerate=30 ! \
  sdxpedestriandetect ! \
  videoscale ! \
  video/x-raw,format=BGR,width=1280,height=720 ! \
  kmssink driver-name=xlrx sync=false

```

You should see output from the webcam piped to the display again, with human figures clearly outlined by bounding boxes. Success!

10 Summary

In this lab you built a complete, end-to-end system incorporating machine learning, computer vision processing acceleration, sensor input, and display. You have seen how the SDSoC tool allows you to easily build complex topologies by leveraging existing, open-source frameworks and seamlessly integrating them with hardware-accelerated functions in the programmable logic.

You created a shared library containing the DPU IP along with some additional support functions to perform color space conversion. The DPU is a versatile processing engine; neural network models and trained weights can be swapped at runtime or even run in parallel on the same DPU instance - swapping our DenseBox-based face detection for another algorithm (for instance YOLO, resnet50, or others) is a straightforward software change.

You then wrapped the accelerated functions as gstreamer plugins, allowing you to build flexible, integrated processing chains leveraging a combination of hardware and software. GStreamer is of course not required, but provides an often-familiar and extremely flexible open-source framework for building flexible, adaptable designs.

Machine learning is a technology rapidly transforming nearly every industry, at scales from the cloud to the edge. It doesn't, however, exist in a vacuum - data must be pre-processed, transformed, and post-processed to take advantage of neural network acceleration, and that processing is often computationally expensive with traditional CPU processing, especially in embedded devices. Xilinx SDx tools are uniquely suited to the task of allowing software developers to create the next generation of adaptable systems, iterating quickly to meet emerging design requirements.