Vision Application using Vitis Libraries

This lab walks you through the steps to create a Vitis design with two kernel **image resize** and **image resize & blur** leveraging the Vitis Accelerated Libraries. Check out Libraries documentation.

Objectives

After completing this lab, you will be able to:

- Use Vitis in command line
- Compile kernels to Xilinx objects (`*.xo`)
- Build FPGA binary file from *.xo files
- Compile host code to object files (`*.o`)
- Build executable host code, which includes linking `*.o` files with dynamic libraries

Steps

Get Vitis Accelerated Libraries repository

Clone the open source Vitis Accelerated Libraries repository in your home directory

```
git clone https://github.com/Xilinx/Vitis_Libraries.git ~
```

Build FPGA binary file

1. Navigate to the root directory of this example

```
cd sources/vision_lab
```

1. Set environmental variables

   It is crucial to define `XCL_EMULATION_MODE` . Let's start with `sw_emu` . We also need to define the target platform, this will be stored in the variable `PFM` , update this path appropriately. Finally, we are going to set up `VITIS_LIBS` . This variabe will point to the `include` directory of vision library L1.

```
export XCL_EMULATION_MODE=<sw_emu|hw_emu|hw>
export PFM=~/aws-fpga/Vitis/aws_platform/xilinx_aws-vu9p-f1_shell-v04261818_201920_1/xilinx_aws-vu9p-f1_shell-v0426
export VITIS_LIBS=~/Vitis_Libraries/vision/L1/include
```

2. Create a temporary folder

   We will use `compile_output` folder to store temporary files to avoid polluting the root directory, this step is not extrictly necessary as the tool will create the folder if it does not exist

   ```
   mkdir compile_output
   ```

3. Compile kernel and build FPGA binary

   In this step the Vitis compiler (`v++`) will be three times:

   - Synthesize the resize kernel (`resize_accel_rgb`) to create a Xilinx object (`compile_output/resize_rgb.xo`)
   - Synthesize the resize & blur kernel (`resize_blur_rgb`) to create a Xilinx object (compile_output/resize_blur.xo)
   - Build the FPGA binary (`vision_example.xclbin`) file

```
#Compile Kernels
v++ --platform $PFM --target $XCL_EMULATION_MODE --log_dir compile_output -c --config src/vision_config.ini -k resize
v++ --platform $PFM --target $XCL_EMULATION_MODE --log_dir compile_output -c --config src/vision_config.ini -k resize
#Create FPGA binary
v++ --platform $PFM --target $XCL_EMULATION_MODE --log_dir compile_output -l --config src/connectivity_aws.ini --conf
```

`v++` is the Vitis compiler command. `v++` compiles and link FPGA binaries, here some of the switches are described. For more information visit Vitis compiler, or run `v++ -h`

- `--platform` or `-f` : specify a Platform
- `--target` or `-t` : specify a compile target: `[sw_emu|hw_emu|hw]` if not defined default is `hw`
- `-c` or `--compile` : run a compile mode, generate a `xo` file
- `-l` or `--link` : run a link mode, generate a `*.xclbin` file
- `-k` or `--kernel` : specify a kernel to compile or link. Only one `-k` option is allowed per `v++` command
- `-g` or `--debug` : generate code for debugging
- `-I` or `--include` : add the directory to the list of directories to be searched for header files. This option is passed to the openCL preprocessor
- `-o` or `--output` : set output file name. Default: a.xclbin for link and build, a.xo for compile
- `--log_dir` : specify a directory to copy internally generated log files to
- `--config` : configuration file
- `--jobs` or `-j` : set the number of jobs

  Note: when compiling for hardware you need to define `__SDSVHLS__` macro, which is used to guard certain part of the code. In this example we need to define `HLS_NO_XIL_FPO_LIB` macro as well, this macro disables the use of bit-accurate, floating-point simulation models, instead using the faster (although not bit-accurate) implementation from your local system. Both macros are included in vision_config.ini file. This macros can be also passed to `v++` as `-D__SDSVHLS__ -DHLS_NO_XIL_FPO_LIB`.

Build host code

In order to build the host application we are going to use `xcpp` a GCC-compatible compiler.

1. Create object files `*.o` of every `*.cpp` file of the host code

    ```
    export XCPP_FLAGS="-c -std=c++14 -g -D__USE_XOPEN2K8 -I$XILINX_XRT/include/ -I$XILINX_VIVADO/include"
    xcpp $XCPP_FLAGS -o compile_output/opencv_example.o ./src/sw/opencv_example.cpp
    xcpp $XCPP_FLAGS -o compile_output/event_timer.o src/sw/event_timer.cpp
    xcpp $XCPP_FLAGS -o compile_output/xcl2.o src/sw/xcl2.cpp
    xcpp $XCPP_FLAGS -o compile_output/xilinx_ocl_helper.o src/sw/xilinx_ocl_helper.cpp
    ```

2. Create executable file linking object files with dynamic libraries

    ```
    export OBJECT_FILES="compile_output/opencv_example.o compile_output/event_timer.o compile_output/xcl2.o compile_out
    xcpp -o vision_example $OBJECT_FILES -lxilinxopencl -lpthread -lrt -lstdc++ -lhlsmc++-GCC46 -lgmp -lmpfr -lIp_float
    ```

Execute the kernels (emulation only)

1. Set `LD_LIBRARY_PATH`

    Make sure you set `LD_LIBRARY_PATH` to include this two path, otherwise the execution will fail.

    ```
    export LD_LIBRARY_PATH=/opt/xilinx/xrt/lib:$XILINX_VIVADO/lnx64/tools/opencv/opencv_gcc
    ```

2. If your are running either `sw_emu` or `hw_emu` an emulated platform needs to be created to simulate the hardware. Skip this step if you are running `hw`

    ```
    emconfigutil --platform $PFM --nd 1
    ```

    This will create a `emconfig.json` file which contains useful information for the simulation tools

3. Run the resize kernel **emulation only**, this may take some time as the hardware is emulated

```
# Run resize kernel with a resize factor of 3
./vision_example vision_example.xclbin 0 src/data/fish_wallpaper.jpg 3
```

You can run `./vision_example` to check the arguments or see the list below

- First argument is the FPGA binary file
- Second argument is the algorithm, `0` resize, `1` resize & blur
- Third argument is the image
- The last argument is the resize factor, you can used integers between 1 to 7, the program will give an error if the output image cannot be generated. Bear in mind, both input and output images size (width and height) must be multiple of 8

  When running `resize` the program will generate two output files: **resize_sw.png** (software execution of the algorithm) and **resize_hw.png** (kernel execution output).
  When running `resize & blur` the program will generate two output files **resize_blur_sw.png** and **resize_blur_hw.png**. View the output files by double-clicking on each of them in File Explorer under `sources/vision_lab/` directory. Compare that to the source input file, **fish_wallpaper.png** located at `sources/vision_lab/src/data`.

Run the kernels in (hardware only)

Since compilation for hardware target will take long time, the FPGA binary is provided in the solution directory. Please follow these steps if you want to generate the FPGA binary outside the workshop/webinar environment

Do Not execute detailed steps 1 to 3 in the workshop/webinar environment. They are instructions for generating FPGA binary

1. The previous compilation was for `sw_emu`. It needs to be set to `hw` for generating FPGA binary. Execute the following command to prepare for the hardware targeting

   `export XCL_EMULATION_MODE=hw`

2. Compile kernels and FPGA binary by repeating step 4 of Build FPGA binary file. The process takes about two hours to complete, generating FPGA binary file called `vision_example.xclbin`.

3. Note: to run in AWS you need to create an AFI. Therefore, the FPFA binary file will be `vision_example.awsxclbin`. In a tutorial, this file will be provided

4. As the FPGA binary is provided, first unset the `XCL_EMULATION_MODE` environment variable by executing

```
unset XCL_EMULATION_MODE
```

5. The host code **does not** need to be recompiled because it **does not** depend on the emulation mode. `XRT` is going to link the executable with the actual hardware instead of the emulated platform.

6. Copy provided solution and execute the following commands to run in hardware

```
# Copy precompiled solution
cp ~/compute_acceleration/solutions/vision_lab/vision_example.awsxclbin
# Run resize kernel with a resize factor of 3
./vision_example vision_example.awsxclbin 0 src/data/fish_wallpaper.jpg 3
# Run resize & blur kernel with a resize factor of 4
./vision_example vision_example.awsxclbin 1 src/data/fish_wallpaper.jpg 4
```

The host application `vision_example` will execute, programming the FPGA and running the host code. This will generate four output files: **resize_sw.png**, **resize_hw.png**, **resize_blur_sw.png** and **resize_blur_hw.png** very quickly compared to `sw_emu` execution done before. Also `profile_summary.csv` and `timeline_trace.csv` files will be generated. These files can be analyzed using `vitis_analyzer`

Clean Directory for a new Build

Before changing the target it is a good idea to clean all files.

```
rm -rf compile_output/ _x/ *.log *.pb *xclbin* *.json *.png *.csv *.exe
```

Conclusion

In this lab, you used Vitis from the command line (`v++`) to create an FPGA binary file with two kernels. You also used `xcpp` to compile the host application. You performed software emulation and analyzed the output images. You then run the provided solution in hardware and evaluate the output images.

Continue to the PYNQ lab

Appendix: exploring the FPGA binary file

We can explore an FPGA binary file (`xclbin`) to obtain design information using `xclbinutil` , which reports xclbin content. For more information about `xclbinutil` run `xclbinutil -h`

Run the following command to save `vision_example.xclbin` content in `vision_example_xclbin.info`

```
xclbinutil --info --input vision_example.xclbin > vision_example_xclbin.info
```

With a text editor open `vision_example_xclbin.info` file, look for the kernel section.

Skip lines until you find Kernel information. Note that the `Signature` is the function prototype. There are only three ports because we used `bundle = control` for the scalar arguments. If you look at the `Instance` section, you will notice that both `image_in` and `image_out` are connected to the `bank 0 (DDR4)` as described in the connectivity_aws.ini file. There is an optimization opportunity in that regard. Observe how the scalar arguments are mapped to `S_AXI_CONTROL` and their offset. You can find a snapshot of `vision_example.xclbin` content below.

```
Kernel: resize_blur_rgb

Definition
----------
   Signature: resize_blur_rgb (ap_uint<512>* image_in, ap_uint<512>* image_out,
     int width_in, int height_in, int width_out, int height_out, float sigma)

Ports
-----
   Port:          M_AXI_IMAGE_IN_GMEM
   Mode:          master
   Range (bytes): 0xFFFFFFFF
   Data Width:    512 bits
   Port Type:     addressable

   Port:          M_AXI_IMAGE_OUT_GMEM
   Mode:          master
   Range (bytes): 0xFFFFFFFF
   Data Width:    512 bits
   Port Type:     addressable

   Port:          S_AXI_CONTROL
   Mode:          slave
   Range (bytes): 0x1000
   Data Width:    32 bits
   Port Type:     addressable

--------------------------
Instance:        resize_blur_rgb_1
   Base Address: 0x10000

   Argument:          image_in
   Register Offset:   0x10
   Port:              M_AXI_IMAGE_IN_GMEM
   Memory:            bank0 (MEM_DDR4)

   Argument:          image_out
   Register Offset:   0x1C
   Port:              M_AXI_IMAGE_OUT_GMEM
   Memory:            bank0 (MEM_DDR4)

   Argument:          width_in
   Register Offset:   0x28
   Port:              S_AXI_CONTROL
```

```
    Memory:             <not applicable>

    Argument:           height_in
    Register Offset:    0x30
    Port:               S_AXI_CONTROL
    Memory:             <not applicable>

    Argument:           width_out
    Register Offset:    0x38
    Port:               S_AXI_CONTROL
    Memory:             <not applicable>

    Argument:           height_out
    Register Offset:    0x40
    Port:               S_AXI_CONTROL
    Memory:             <not applicable>

    Argument:           sigma
    Register Offset:    0x48
    Port:               S_AXI_CONTROL
    Memory:             <not applicable>
```