ECE 5727

Lab 5: HLS - Video Processing

Due Wednesday, May 12
Total Points: 35

Objectives

In this lab you will:

1. Create a real-time video processing pipeline that uses a Sobel filter to highlight edges in the video
2. Work through the process of creating an algorithm using an HLS design flow, and adding the resulting IP core to an FPGA design

Lab Overview

High-level Synthesis (HLS) is the usage of normal programming languages (C, C++, MATLAB) to describe a desired algorithm from which an RTL design can be extracted. This allows the designer greater abstraction and ease of implementation of complex algorithms, such as those used in image processing, than HDLs (VHDL, Verilog, SystemVerilog) can provide.

Xilix and Digilent partnered to create an example of using HLS to implement video processing on their Zynq chip. For this lab, you will be working through this example to gain a high-level understanding of how HLS is used, and how it differs from normal C programming.

Instructions

Setup

1. Download the lab files from the course Canvas or GitHub.
2. Make sure you have Vitis HLS installed. It should have been an option when you originally installed Vivado, but if you don't have it:
   - Linux: Launch `Vivado > Help > Add Design Tools or Devices`
   - Windows: `Start > Xilinx Design Tools > Add Design Tools or Devices`

   Then go through the installation/upgrade procedure being sure to include HLS.
3. When Xilinx switched from Vivado HLS to the new Vitis HLS platform, to reduce its footprint, they eliminated a lot of their libraries from the distribution. Instead, the new libraries must now be downloaded from their GitHub: https://github.com/Xilinx/Vitis_Libraries. Clone or download this to somewhere on your computer. We will only need the `vision` directory within it, so you can keep just that if you want.

4. As part of this elimination of the vision (image/video processing) libraries from the distribution, OpenCV is also no longer part of the platform binary. So that will need to be installed as well. The libraries have been verified to work with 3.x, though you can use 4.x if you want and likely be fine.

    - Linux users can follow the instructions here: https://docs.opencv.org/master/d7/d9f/tutorial_linux_install.html. I believe you can just download the pre-built libraries or download the source and build it fresh.
    - Windows users should follow the instructions here: https://medium.com/csmadeeasy/opencv-c-installation-on-windows-with-mingw-c0fc1499f39. Do NOT just download the pre-compiled version from OpenCV's sourceforge or follow their instructions as either will result in libraries compiled for VisualStudio which will not work with Vitis. Compiling the libraries is going to take a *very* long time. So go do something else for an hour.

Create Vivado Project

Workbook.pdf contains instructions with images of each step necessary. Note however, those instructions were created with an older version of Vivado and Vivado HLS, not Vitis HLS. This means some instructions will not be correct, so when in doubt, refer to my instructions which reflect doing all this on Vivado/Vitis 2020.2. I have also attempted to include a little more information in my version of the instructions.

Parts 1-3 of *Workbook.pdf* are an overview of FPGAs and HLS. You can read this at your leisure. Part 4 uses the example HLS project. You can skip this, as the remainder of the Lab will take you through everything this covers in more detail.

Part 5 of the Workbook sets up the baseline projects for Vivado and Vitis SDK. These create a pass-through video pipeline. You can skip this if you'd like but if not:

1. Launch `Vivado > Tools > Run Tcl Script > Lab5/hw/proj/create_project.tcl` and run it. This will generate the complete Vivado project and populate the board design.

2. Unlike the Workbook, this block design has a fourth hierarchy block (`video_proc`). Expanding that block will reveal four main IP blocks. Two are HLS created blocks Digilent created that act upon the video stream. The other two are AXI-Switches, which is what allow the processor to direct the video stream through the desired video processing IP (or bypass them entirely). The switches have four available connections, and only three are currently in use. This lab is to create a third processing block that will hook into the unused slot.

3. I have included an exported Vitis SDK project (the `.ide.zip` file). If you have access to the hardware, as well as an HDMI cord and tv/monitor, you can import this file into Vitis SDK which will allow you to run the design on the board. Let me know if you want to do this and I can give you further instructions on doing all that if you need it.

Edge Detection HLS

Create HLS Project

It is now time to create our video processing HLS core. The project setup instructions are completely different for Vitis from Vivado, so ignore the Workbook entirely.

1. Launch `Vitis HLS > Create New Project`
2. Name the project `edge_detect` and select `Lab5/hls_project` for the location.
3. `Add Files... > Lab5/hls_project/edge_detection.{h,cpp}`
    - Browse for Top Function, and select the only option: `edge_detect`
    - Select `edge_dection.cpp` and then click `Edit CFLAGS...` Add the following: `-I<path to Vitis_Libraries>/vision/L1/include`
4. Add testbench files `Lab5/hls_project/{edge_detection_test.cpp, fox.bmp}`. Select the cpp file and add the same CFLAG as above. Also add the following:
    - Linux:
    - Windows: `-I<path to opencv>/install/include`
5. For solution configuration set the Clock Period to 6.67 (so it builds for a 150MHz clock), and select `xc7z010clg400-1` for Part Selection. It can help to filter by `Package: clg400` to find it quickly. The flow should be left at its default of Vivado IP.
6. Once the project is open, we need to make sure we can link the OpenCV libraries for simulation so select `Project > Project Settings > Simulation`. Within the Linker Flags box put the following:
    - Linux: `-L <path to opencv>/tbd -lopencv_imgcodecs -lopencv_imgproc -lopencv_core`
    - Windows: `-L <path to opencv>/install/x64/mingw/lib -lopencv_imgcodecs3414 -lopencv_imgproc3414 -lopencv_core3414` Notice that unlike Linux, for Windows the libraries have the version number appended. So these are for building with OpenCV 3.4.14. If you built with a different version, update accordingly.

Solution 1

Unlike with Vivado HLS, Vitis HLS now includes interface type checks and so will fail if we attempt to simulate/compile the code as it exists. This is because we have not specified what type of interfaces we want for our `stream_in` and `stream_out_{x,y}` parameters. We will also go ahead and specify the control interface we want to use for our IP block at this time. This is all done using `Directives`, which can either be specified directly within

the code or as part of a `directives.tcl` script. I find it is best to put constant directives directly into the file, and exploratory ones into the script. We use "solutions" to explore the different ways to generate our RTL, and each solution will use the same code but a different directives script. So, by putting things like the interface directives within the code, that will automatically apply them to all solutions. The GUI does allow you to copy the directives script from a solution when creating a new one, but this still requires syncing all the scripts if you want to change the interface and removing unnecessary directives. So for solution 1, which will have no additional directives beyond interfaces, we will be putting all of them directly into the code.

These directions will roughly match those of the Workbook beginning at the bottom of page 33. Skip the "creating new solution" step at the top of the page.

1. Open `edge_detection.cpp` and on the right side select the `Directive` tab.
2. Select the `stream_in` and `stream_out_{x,y}` interfaces. Right-click the selection and choose `Insert Directive`.
   - `Directive: INTERFACE`
   - `Destination: Source File` This will put the directives into the cpp file instead of the solution1 directives.tcl script.
   - `mode: axis` This specifies the interface is an AXI-Stream.

   Upon clicking OK, you should see three `#pragma`s appear at the top of the function.
3. Right-click the function `edge_detect` and again do an INTERFACE directive. This time set the mode to `ap_ctrl_none`. This means we want no control signals to the block, so it will run whenever it receives data on the stream. The other processing blocks within the FPGA design from Digilent are set with (I believe) the `ap_ctrl_chain` interface which means it will store data from the stream until a start signal is asserted into the block.
4. Do NOT apply the DATAFLOW directive as the Workbook states. That will be for our solution2.
5. Save the changes to the cpp file. Our HLS algorithm is all set to go now! Choose `Project > Run C Simulation`. This will take a minute or two to run. If it fails due to an error, most likely it is an issue with including and/or linking one of the libraries. Once it is done you will see there is a new *csim* folder under the solution1 in the explorer. Open a file explorer window and navigate to the *csim/build* directory. There will be a bunch of .bmp files. `fox.bmp` is the original input image. The ones with `cvsobel` are created using opencv. The ones with just `sobel` are created by the HLS algorithm. Hopefully they look pretty similar!
6. It is now time to generate RTL. Select the green arrow from the toolbar or via `Solution > Run C Synthesis`. Again, this will take a minute or two. Once it is done running, a report should pop up with information about the generated design.

Most important is the Timing Estimate, which tells us in this case that our design will be able to run with our targeted 150MHz clock, since it estimates the slowest logic to take only 4.869ns. Under the performance section we can see information on how long the actual processing will take to process a single 720x1280p frame of video by looking at the Latency column. Here we can see that each part of the algorithm takes approximately 6ms to run, meaning the entire process takes about 37ms. This means we wouldn't be able to exceed a frame-rate of about 27fps (1000ms / 36ms per frame). This is pretty slow so lets see if we can't direct the compiler to a faster solution.

Solution 2

We want to keep our solution1 results as a baseline to compare against, so we need to create a new solution to experiment with. Choose `Project > New Solution...` Everything should pre-populate, and you can choose to copy from solution1 or not. Since we put all the interface directives into the code, we don't need to copy, but if there were things within the directives.tcl script we wanted to keep, this is what that would be for. Once the solution is created it becomes "Active". This is denoted within the Explorer tab by bolding it. Simulation, synthesis, Add Directive etc commands are run against the active solution.

1. Now is when we want to add the DATAFLOW directive to our design. Do so by right-clicking the `edge_detect` function in the Directive tab to Add Directive. This time, we want the directive to go into the script (Directive File), since it is only to be applied to this solution.

2. You can run simulation again if you want, but the results won't differ since we haven't functionally changed the design.

3. Run C Synthesis. This time it will do so with the new DATAFLOW directive, which tells the compiler our function is a series of steps in a datapath and it can pipeline the data through it. This means it can treat each sub-part of the function as an individual function and can start a frame into it as soon as the prior one has cleared.

4. When synthesis has finished, a new report will pop up. We want to compare to our solution1 results so in the upper right corner of the GUI choose to go to `Analysis`. This page allows us to dig deeper into the results from any of our solutions. If a report comparison does not pop up automatically, choose `Project > Compare Reports` and then choose our two solutions.

5. Looking at the comparison, we can see that the Latency has dropped from 37ms in solution1 to 6.2ms in solution2. This is somewhat misleading, because what really dropped was the Iteration Interval, which is how long has to pass before the algorithm can be started again (a new frame passed in). The latency *should* reflect how long it takes a single frame to pass through the entire algorithm which should still be 37ms. Vivado HLS did this as well and its a weird bug for them to have not fixed. Regardless, what matters is that the Interval has dropped to 6ms, meaning our algorithm can now function at a much higher frame rate.

6. We are now happy with the theoretical performance of our block, so it is time to check the RTL generated functions correctly. This is done by running a co-simulation. You can switch the GUI back from Analysis to Synthesis, and make sure solution2 is still the active solution. Choose `Solution > Run C/RTL Cosimulation`. This will take 5-10 minutes, and there are times it will seem like its frozen/not doing anything. Don't worry, it's running. Eventually it will finish and you can check the simulation results just like you did for the C simulation by looking at the generated images. This time they are located under `solution2/sim/wrapc_pc`

Create IP Core

At this point, the lab is effectively over. To add the core to the FPGA design you would choose `Solution > Export RTL`. This creates all the IP Core files and exports them in a zip file. Extracting that into the `Lab5/hw/repo/local` folder would add it to the IP catalog and you'd then be able to drag it into the block design and connect it to the AXI Switches. Exporting the core also gives you a chance to run synthesis (or synthesis and implementation which is weird to me) which would get you more accurate utilization numbers for the core to look at in the HLS GUI before you actually add it to your design in Vivado.

Lab Questions

Turn in answers to the below questions, as well as

1. (*15 points*) Take a look at the analysis for the HLS Solution 2.
   (a) (*5 points*) What is the iteration interval (II) of the algorithm? This is the number of clock cycles that must elapse between inputs. How much of an improvement is this over Solution 1?
   (b) (*5 points*) With a 100 MHz clock, what is the maximum frame rate (fps) we can handle?
   (c) (*5 points*) Which part of the algorithm appears to be the limiting factor for latency and II?
2. (*10 points*) Take a look at the comparison of Solutions 1 and 2. Adding the DATAFLOW directive from solution 1 to solution 2 improved our iteration interval.
   (a) (*5 points*) What cost do we incur by using this directive? (Take a look at the utilization numbers)
   (b) (*5 points*) What do you think is causing this?
3. (*5 points*) Turn in either x or y C simulation output images.
4. (*5 points*) Turn in either x or y co-simulation output images.