

EE216 Re-configurable Computing Homework7 Report

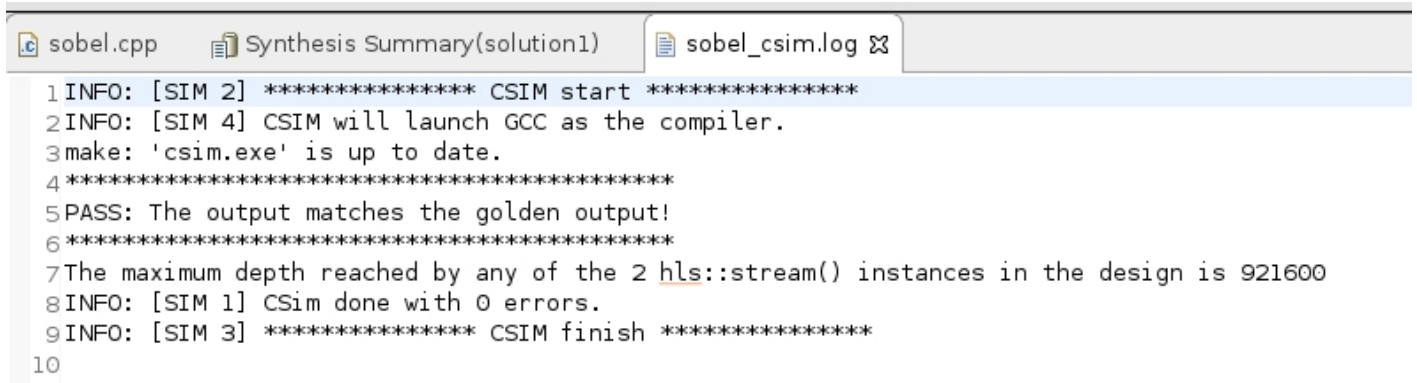
Zhaojun Ni, 2022231102

December 23, 2023

1 Sobel HLS IP Block

After the new testbench for homework 6 was released, I didn't pay much attention to it since I had already submitted my homework. However, when working on this homework later, I found deficiencies in the original code, mainly related to data types and bit widths. The original data type I set was `ap_uint9`, and at that time, the testbench did not encounter any errors. The new testbench has inputs and outputs of 8-bit unsigned char type, but simulation errors occurred because the intermediate `gradient_x` and `gradient_y` did not limit the convolution results to the range of 0 to 255. When calculating the gradients separately, it is necessary to consider the possibility of data overflow after multiplication and addition. Therefore, I changed the data type of the intermediate values to `ap_int11`, represented by `TMP`. The range method can be used to index the bits, so for int data, if the highest bit is 1, it is negative, and the output is 0. Otherwise, check if the data is greater than 255; if so, truncate the data to 255. The final summation is defined as `ap_uint9`. Since the data to be added are all non-negative numbers, it is only necessary to consider whether there is overflow in the ninth bit after the summation. If the ninth bit is 1, it indicates overflow, and the output is 255. Otherwise, if there is no overflow, the output is the value of the lower 8 bits.

Before exporting the IP, it is necessary to conduct C simulation, synthesis, and co-simulation. The exported IP is a zip file.



```
sobel.cpp  Synthesis Summary(solution1)  sobel_csim.log x
1 INFO: [SIM 2] ***** CSIM start *****
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3 make: 'csim.exe' is up to date.
4 *****
5 PASS: The output matches the golden output!
6 *****
7 The maximum depth reached by any of the 2 hls::stream() instances in the design is 921600
8 INFO: [SIM 1] CSim done with 0 errors.
9 INFO: [SIM 3] ***** CSIM finish *****
10
```

Figure 1: CSIM output matches the golden output.

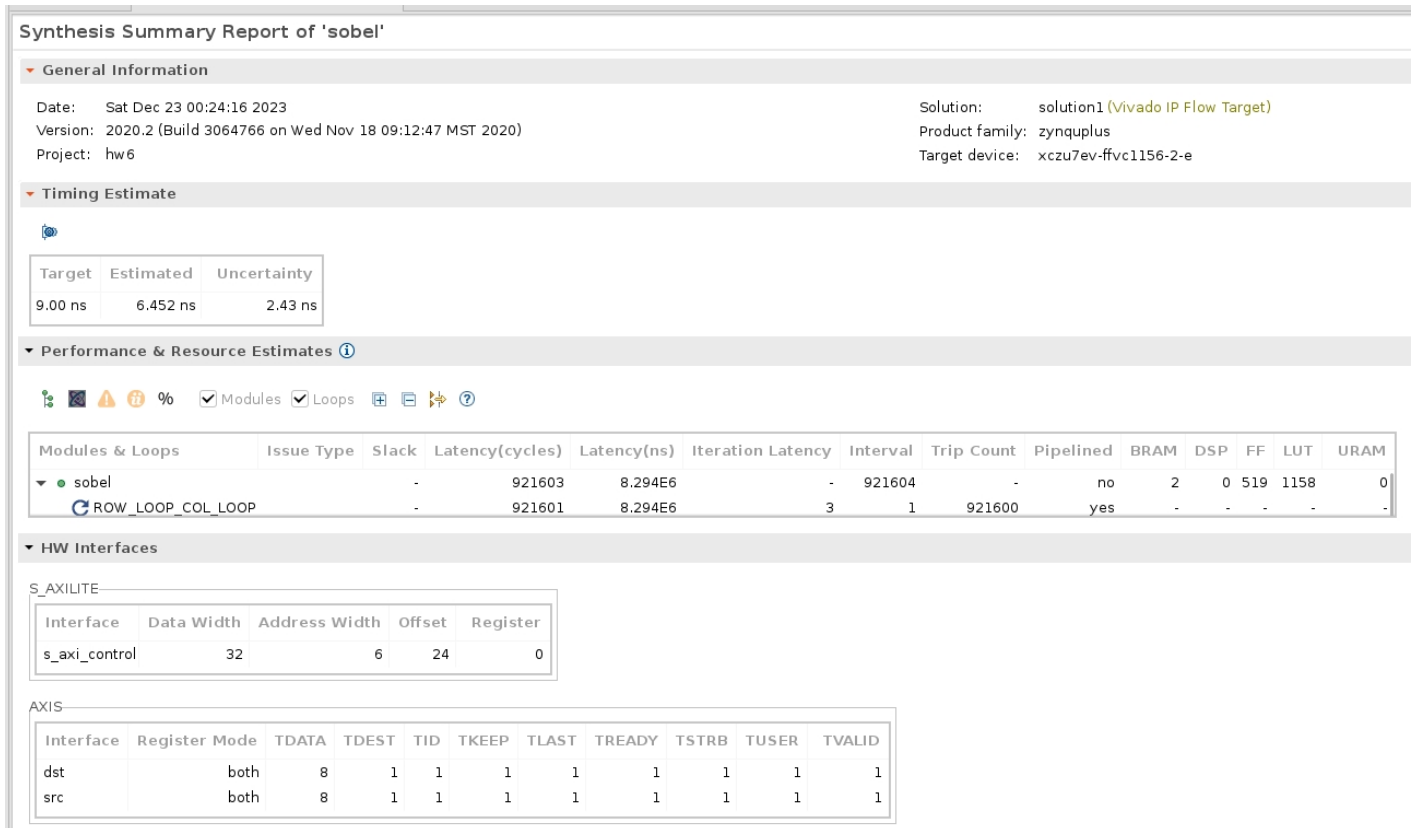


Figure 2: Synthesis Report

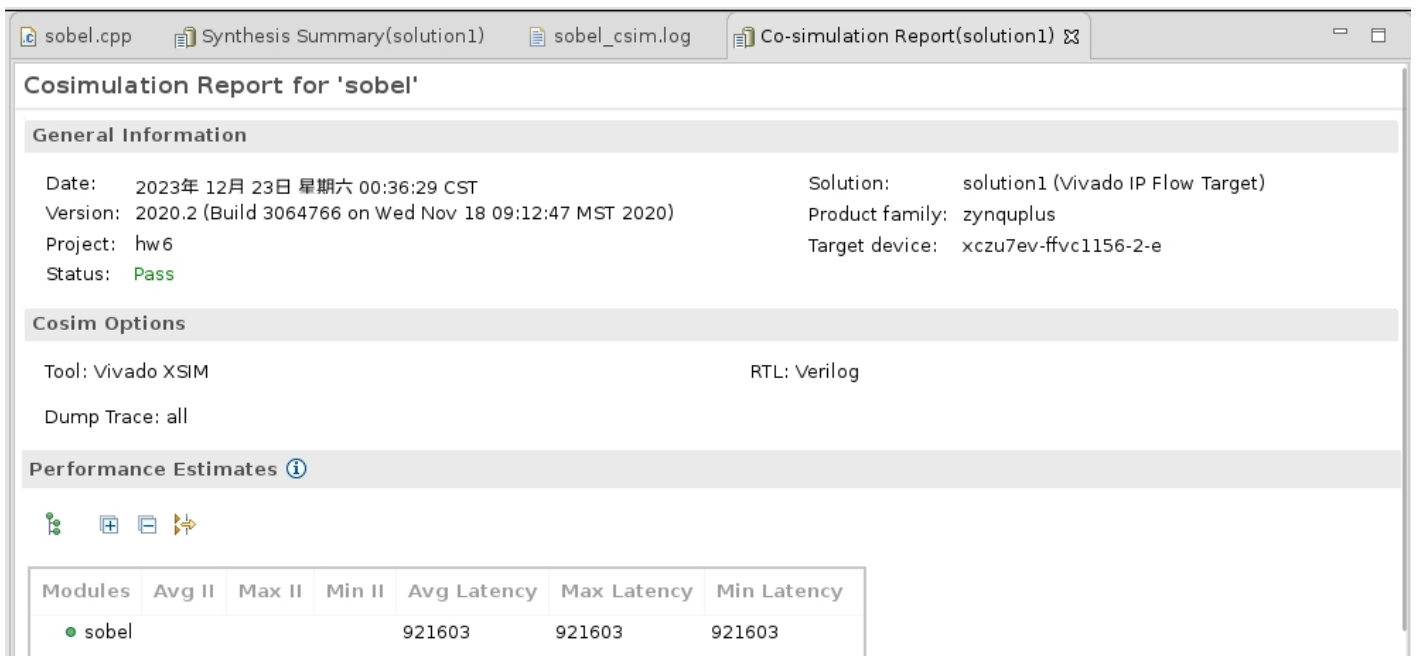


Figure 3: Cosimulation passed.

2 Create Block Design and Generate Bitstream

In Vivado, to make the PS and PL work together, they need to be connected in the block design. First, import the Sobel IP created in Vitis HLS, then create a block design, select Zynq UltraScale+ MPSoC, and configure the PS-PL interfaces. Since HLS Stream is used, the setup process is similar to the method used in Lab7. Communication between the IP and PS is achieved through AXI DMA, and since the data type being sent is 8 bits, the Stream Data Width needs to be set to 8. It's important to note that in subsequent experiments on the board, if the **width of the buffer length register** is kept at the default value of 14, the following error may occur.

```
ValueError                                Traceback (most recent call last)
<ipython-input-15-db515cdb3275> in <module>
      1 # Launch the IP and DMA
----> 2 dma.sendchannel.transfer(in_buffer)
      3 dma.recvchannel.transfer(out_buffer)
      4 sobel.write(0x18,row)
      5 sobel.write(0x20,col)

/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq/lib/dma.py in transfer(self, array, start, nbytes)
    176         nbytes = array.nbytes - start
    177         if nbytes > self._max_size:
--> 178             raise ValueError('Transfer size is {} bytes, which exceeds '
    179                             'the maximum DMA buffer size {}'.format(
    180                                 nbytes, self._max_size))

ValueError: Transfer size is 921600 bytes, which exceeds the maximum DMA buffer size 16383.
```

Figure 4: Error caused by insufficient transfer size.

Initially, I didn't know how to resolve this bug. Later, I found that 16383 is 2 to the power of 14 minus 1. This led me to consider that the buffer length might not be set large enough. Since the input is 720*1080, 2 to the power of 20 meets the bit-width requirements. After modifying it to 20, there were no more errors during board testing.

The subsequent steps involve completing the connections in the block design, generating the block design, creating an HDL wrapper for the top-level design, and then generating the bitstream after synthesis. Once the bitstream is generated and exported, locate the hardware handoff. With both files ready, prepare to proceed with the board setup.

3 Write Software Driver

In PYNQ, for agile development and convenient debugging, it is common to use Jupyter Notebook for testing. Once the code runs smoothly, it can be translated into a Python script for further use.

In the terminal, start the Jupyter server by entering the command “jupyter notebook --port=8889 --allow-root”. The terminal will display a URL, such as “http://pynq:8889/”. Next, open a new Jupyter notebook file (ipynb), choose the option to use an existing Jupyter server in the upper right corner, and enter the earlier URL. It's important to note that entering “!pwd” reveals that the current path is not where the ipynb file is located. Therefore, I opted for absolute paths in the code, such as “/root/junjun/...”. The software driver code can be divided into the following steps.

```

from pynq import Overlay, allocate
import numpy as np
import time
import cv2

```

Python

```

WIDTH = 1280
HEIGHT = 720
WIDTHOUT = 1278
HEIGHTOUT = 718

```

Python

```

# Download bitstream
print("programing FPGA")
overlay = Overlay("/root/junjun/sobel.bit")
print("programing FPGA done")

```

Python

```

programing FPGA
programing FPGA done

```

Figure 5: Download bitstream.

```

# Get the IP object from the overlay
sobel = overlay.sobel_0
dma = overlay.axi_dma_0

```

Python

```

# Get input image and golden result
img_in = cv2.imread("/root/junjun/input.png", cv2.IMREAD_GRAYSCALE)
out_ref = cv2.imread("/root/junjun/reference.png", cv2.IMREAD_GRAYSCALE)

```

Python

```

# Allocate buffers
in_buffer = allocate(shape=(WIDTH*HEIGHT,), dtype=np.uint8)
out_buffer = allocate(shape=(WIDTHOUT*HEIGHTOUT,), dtype=np.uint8)

```

Python

```

# Initialize input data
for i in range(HEIGHT):
    for j in range(WIDTH):
        in_buffer[i*WIDTH+j] = img_in[i][j]

```

Python

Figure 6: Get IP objects and image inputs, allocate buffers and initialize input data.

```
print('Testing sobel performance...')
N = 1000
start_time = time.time()
for t in range(N):
    # Launch the IP and DMA
    sobel.write(0x18, HEIGHT)
    sobel.write(0x20, WIDTH)
    sobel.write(0x0, 0x1)
    dma.sendchannel.transfer(in_buffer)
    dma.recvchannel.transfer(out_buffer)
    dma.sendchannel.wait()
    dma.recvchannel.wait()
end_time = time.time()
fps = N / (end_time - start_time)
print("fps =", fps)
```

Python

```
Testing sobel performance...
fps = 104.73731181581233
```

▶ ◀ ☐ ... 🗑

```
# Allocate result and evaluation
result = allocate(shape=(HEIGHTOUT, WIDTHOUT), dtype=np.uint8)
error = 0
for i in range(HEIGHTOUT):
    for j in range(WIDTHOUT):
        result[i][j] = out_buffer[i*WIDTHOUT+j]
        if result[i][j] != out_ref[i][j]:
            error += 1
print(f'The output has {error} errors.')
cv2.imwrite("/root/junjun/result.png", result)
```

Python

```
The output has 0 errors.

True
```

Figure 7: Run 1000 rounds of tests, write control signals to launch sobel IP and DMA. Compare result with golden data.

After completing the code development within the ipynb, it can be written into a .py file. The final execution result is shown in the figure below. The average throughput has reached 104.857 frames per second.

```
● (pynq-venv) root@pynq:~/junjun# python sobel.py
programing FPGA...
programing FPGA done
Testing sobel performance for 1000 rounds...
fps = 104.85725134963926
Evaluating result...
The output has 0 errors.
The result figure is saved.
```

Figure 8: Console output.

The following two images show the input image to be processed and the result image after Sobel filtering has been applied.



Figure 9: Input image.



Figure 10: Output image.