EE6470 ESL Design and Synthesis

HW1 Laplacian of Gaussian filter

110061146 林汶昇

https://github.com/patata0717/EE6470-ESL-design-and-synthesis/tree/main/hw1

1. Introduction

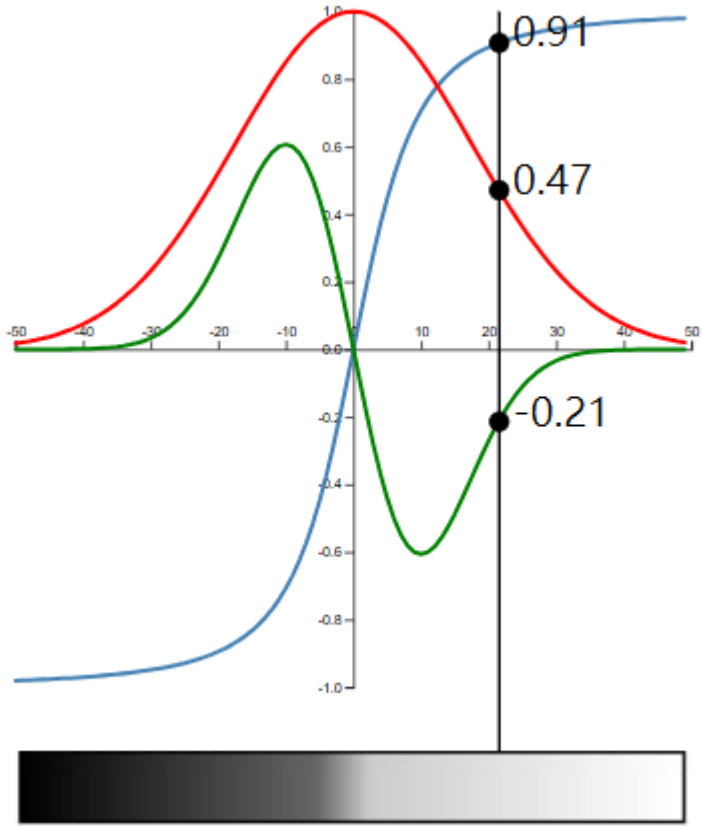
In this lab, we will implement a Laplacian of Gaussian filter. Using hardware and software co-design. And then try to use input data buffering to make it more efficient.

2. LoG filter

LoG filter is a common filter for edge detection, it using Gaussian filter to remove the noise of the image, and then use Laplacian filter to sharpen the edge.

Laplacian filter is a second derivative function, which captures the accelerative transition, for the image below, we can catch where L(x, y) is 0 to detect edge.

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$



Red: sobel Blue: intensity Green: Laplacian

https://cse442-17f.github.io/Sobel-Laplacian-and-Canny-Edge-Detection-Algorithms/

For 3 by 3 filter, it can be approximated to

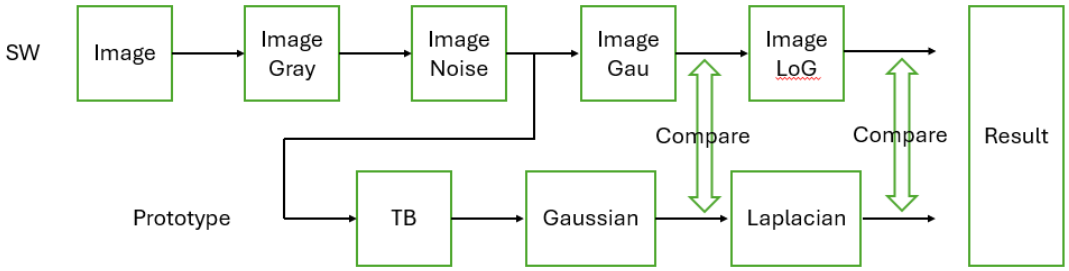
0	-1	0
-1	5	-1
0	-1	0

Because it is second derivative, it is very sensitive to noise, so we will use Gaussian filter to blur the noise point, the Gaussian filter can be approximated to

1	2	1
2	4	2
1	2	1

We will concatenate these 2 filter to be a LoG filter.

3. HW/SW co-design



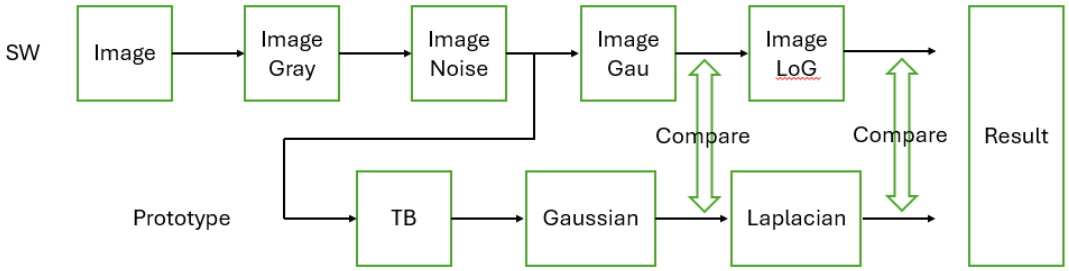
SystemC is a tool for prototyping a design. We develop a prototype for HW/SW co-design. In this homework, we still use SW(python code) as golden to verify systemC's results.

First, SW will convert image to gray scale, for detail, we are using .bmp file, it will have 2 ways to represent gray-colored images:

1. [Colored mode] All RGB set to same value, thus, output will be gray scale.
2. [Gray mode] Use intensity to represent gray color.

Colored mode has 4 channels(R, G, B, ~~transparency~~), Gray mode has only 1 channel. The python code will use method 2. So the image data size is shrink to 1/3. And the channel need to rewrite(R, G, B→Intensity).

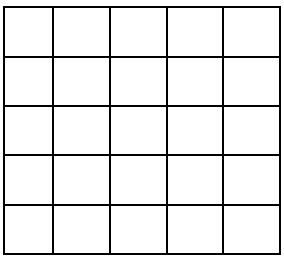
192 KB (197,020 位元組)	65.0 KB (66,614 位元組)



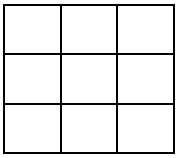
Keep going, and we will manually add noise to image, and then pass it to testbench, and TB will sent value to Gaussian filter, Gaussian filter will sent value to Laplacian filter, and Laplacian filter will sent value to testbench and write it to bitmap. We can compare result with python output golden.

4. Part1: LoG without buffering

Input:



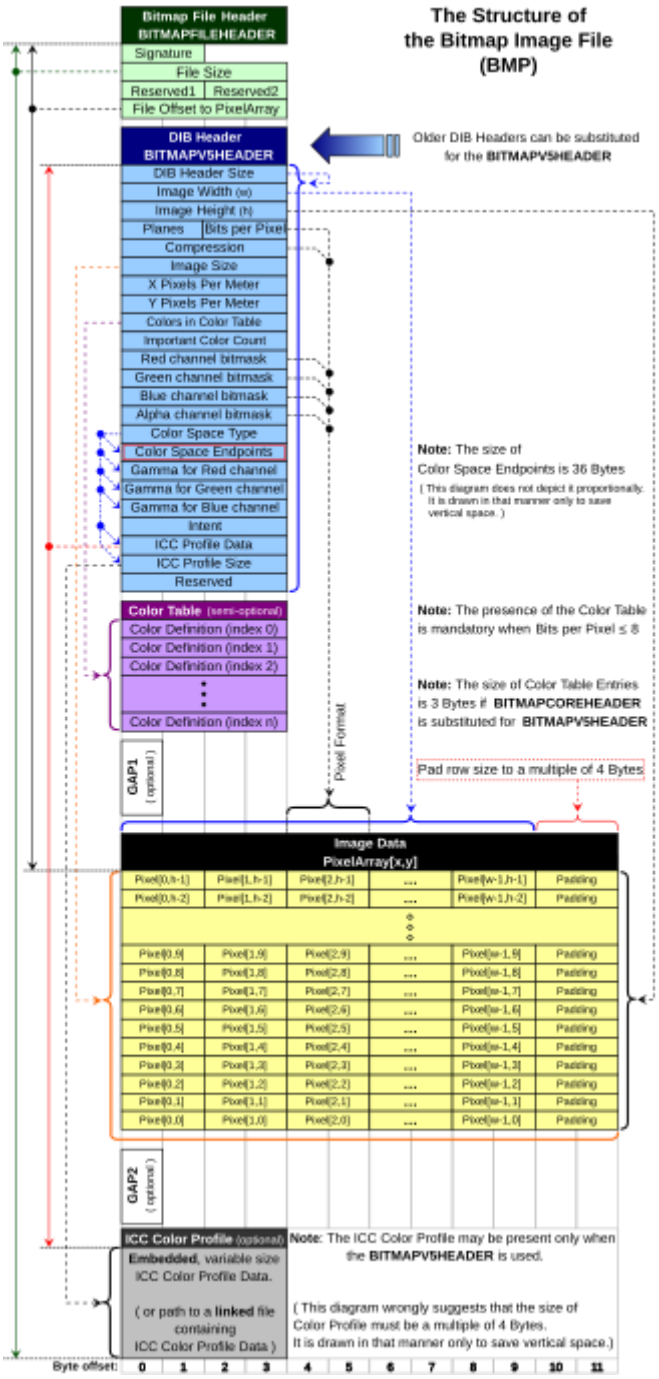
Gaussian:



Laplacian:

For each Laplacian output, it needs the exact location in Gaussian output and it's 3 by 3 neighbors. And recursively, each Gaussian needs input file's 3 by 3 neighbors. So we can see that, with naïve approach, we need 9×9=81 input pixels to compute a LoG output.

Ok, let's talk about the implementation. First we start with code for Sobel filter, but we need to modify it significantly. First, the mode is different, we need to modify RGB mode to Gray mode. This is the bmp file structure



Simply put, .bmp contains a header, plus data for image. We need to modify several arguments in header. Header has 54 char for color mode and 1078 char for gray mode.

```c
unsigned char header[1078] = {
    0x42,           // identity : B
    0x4d,           // identity : M
    0,    0, 0, 0, // file size
    0,    0,        // reserved1
    0,    0,        // reserved2
    54,   4, 0, 0, // RGB data offset
    40,   0, 0, 0, // struct BITMAPINFOHEADER size
    0,    0, 0, 0, // bmp width
    0,    0, 0, 0, // bmp height
    1,    0,        // planes
    8,    0,        // bit per pixel(gray)
    0,    0, 0, 0, // compression
    0,    0, 0, 0, // data size
    0,    0, 0, 0, // h resolution
    0,    0, 0, 0, // v resolution
    0,    0, 0, 0, // used colors
    0,    0, 0, 0  // important colors
};
```

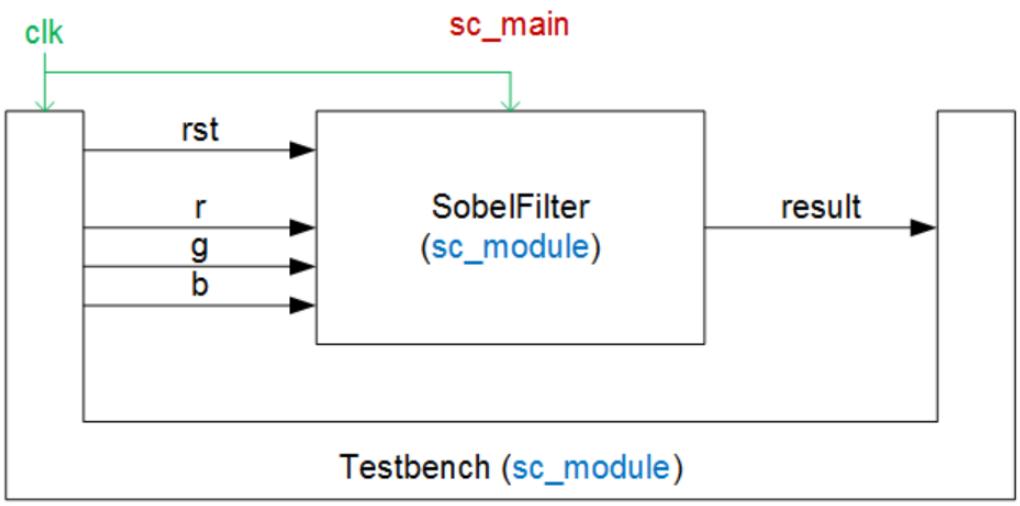
```c
// ** Add grayscale color palette (256 entries) **
for (int i = 0; i < 256; i++) {
    header[54 + i * 4] = i;   // Red
    header[55 + i * 4] = i;   // Green
    header[56 + i * 4] = i;   // Blue
    header[57 + i * 4] = 0;   // Reserved (always 0)
}
```

Changes include:

1. RGB data offset: 54 to 1078(54 + 4*sizeof(char))
2. bmp width, height: assign by code
3. bit per pixel: 24 to 8
4. Add color palette(1024)

Thus, we can use it as gray mode.

Second, the channel is different, Sobel filter is originally RGB, and it now use only 1 channel, intensity, so as Gaussian and Laplacian filter.



And then, we implement naïve approach.

```
for (y = 0; y != height; ++y) {
  for (x = 0; x != width; ++x) {
    // Process a 3x3 block
    for (i = -1; i <= 1; ++i) {
      for (j = -1; j <= 1; ++j) {
        // Process each 3x3 region in the 5x5 block
        for (v = -1; v <= 1; ++v) {
          for (u = -1; u <= 1; ++u) {
            int yi = y + i + v;
            int xi = x + j + u;
            if (((x == 0)          && (j == -1)) ||
                ((x == width - 1)  && (j ==  1)) ||
                ((y == 0)          && (i == -1)) ||
                ((y == height - 1) && (i ==  1))  ) {
              INTENSITY = 0;
            } else if (xi >= 0 && xi < width && yi >= 0 && yi < height) {
              INTENSITY = *(source_bitmap + width * yi + xi);
            } else {
              INTENSITY = 0;
            }
            o_intensity.write(INTENSITY);
```

We can see that it has 2 level of 3 by 3 block, we first process 3 by 3 block for Laplacian, and for each Gaussian 5 by 5 resign, we need to process 3 by 3 9 times. And finally, we need to handle out of bound conditions. Noted that we can't treat out of bound pixels as 0 input, because the filter will still add boundary values. So for Gaussian at out of bound with 1 pixel, we need to assign it as 0 directly. And then we write the intensity value.

We using Cmake to collect input and source file, compile systemC and compare

result with golden.

```
        SystemC 3.0.1-Accellera --- Jan 10 2025 14:04:40
        Copyright (c) 1996-2024 by all Contributors,
        ALL RIGHTS RESERVED
Image width=256, height=256

Info: /OSCI/SystemC: Simulation stopped by user.
Simulated time == 5308416 ns
[100%] Built target run
[ 83%] Built target LoG
[100%] Built target run
Files /home/u110/u110061146/ee6470/hw1/part1/build/out.bmp and ../SW/golden/lena_color_256_gray_noise_gaussian_laplacian2.bmp
are identical
[100%] Built target check
```

For different image input, we can modify argument in Cmake, to generate filter_def.h needed and then compile, it can handle image at different size.

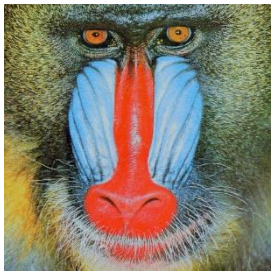
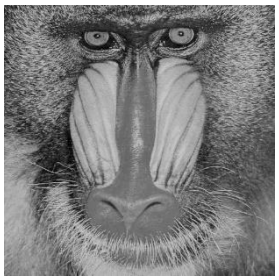
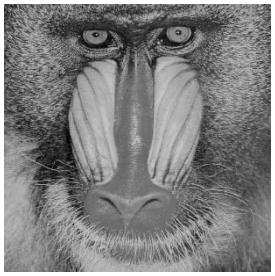
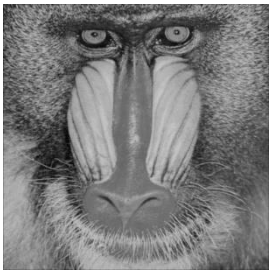
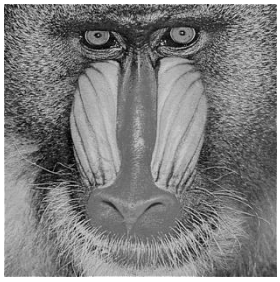
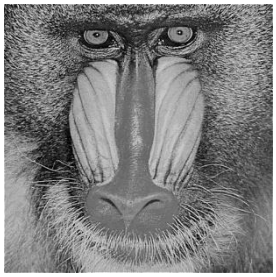
```
# Set file name
set(FILE_NAME "lena_color_256") # modify here
set(INPUT_FILE "${FILE_NAME}_gray_noise.bmp")
set(GOLDEN_FILE "${FILE_NAME}_gray_noise_gaussian_laplacian2.bmp")

# Set image size
set(WIDTH 256)  # modify here
set(HEIGHT 256) # modify here
```

Lena(256x256)

Ori	Gray	Noise
Gaussian golden	LoG golden	Output

Simulated time == 5308416 ns

Mandril(512x512)

		
Ori	Gray	Noise
		
Gaussian golden	LoG golden	Output

Simulated time == 21233664 ns

5. Part2: LoG with input data buffering

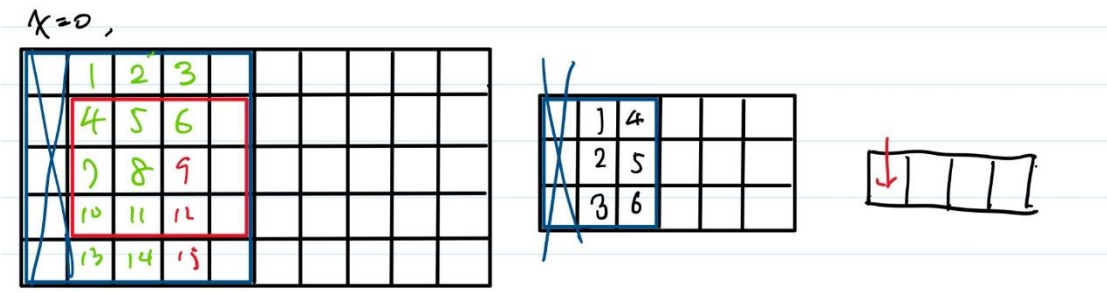
And then, we now implement data buffering.

The max data buffer size is 9, it may use register in real hardware design.

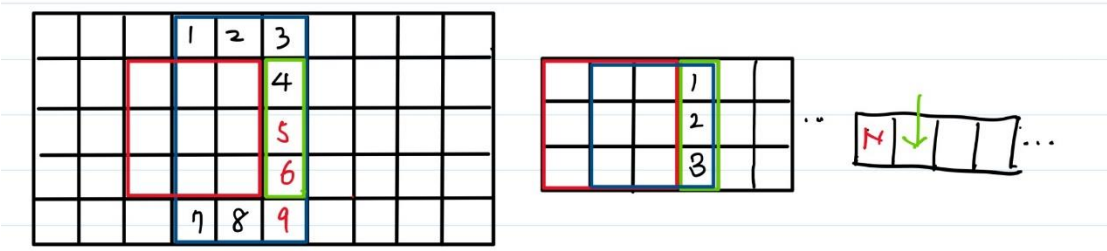
Register is the main area cost for hardware, so we need to minimize it as possible.

To sum up, my method needs 9 pixel buffer for Gaussian, and 6 for Laplacian.

And it can achieve nearly 1/9 of original cycle count.

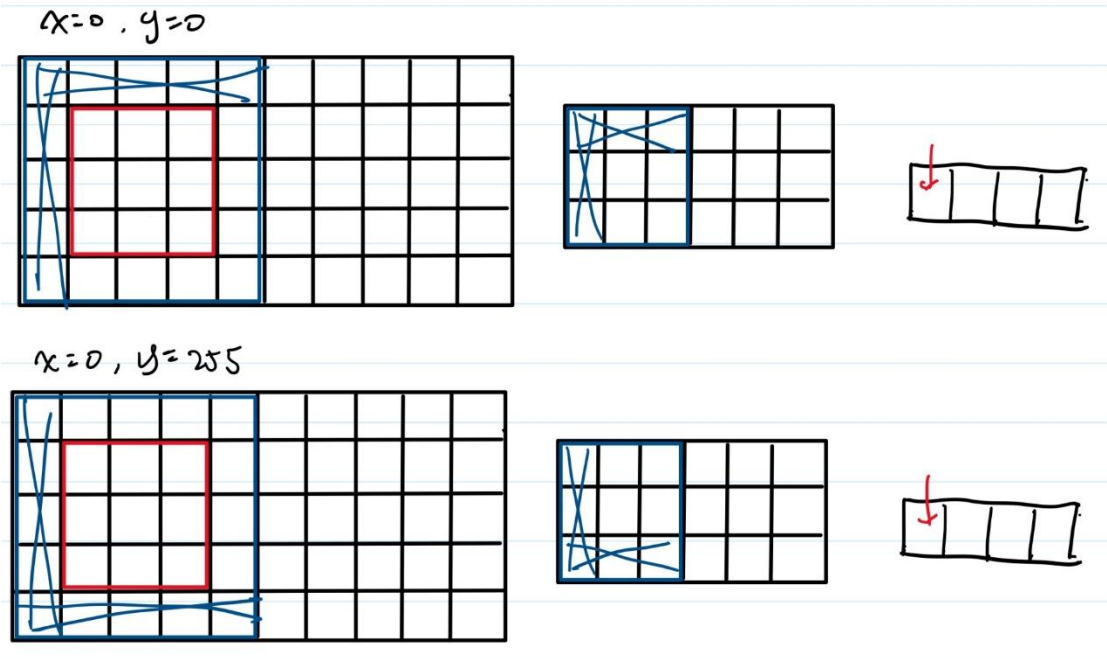


First, at each new row, we need to refill the buffer. The first column is always zero, so we ignore it. The rest we need to ask for 15 data for 3 Gaussian output. Laplacian output is not yet ready. Gaussian will print output at cycle 9, 12, 15.

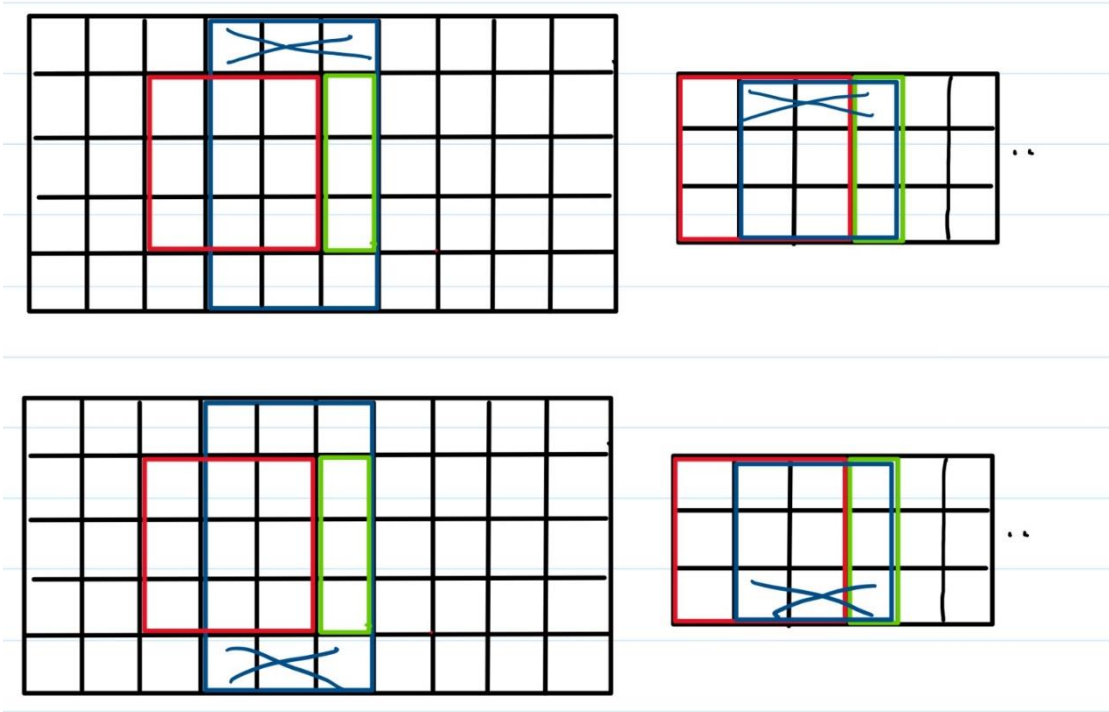


Then, this is the case after first 15 cycles finished. For each 3 Gaussian output, 1 Laplacian output, we need to ask 9 pixels as input. Gaussian will print output at 5, 6, 9.

When a convolution result sent, the new value well immediate convolute with the pixel in the bucket, and then read new input and keep convolute, so there is at most 9 buffer plus 1 value in channel, total 10 values in the filter.



For corner case handling, there is 2-level discussion, When accessing (0, 0), the out-of-bound Gaussian should be zero, it can't convolute with original image, else it will touch the non-zero value and produce incorrect results. We handling this inside Gaussian filter. And for original input, when we accessing out-of-bound data, we directly give it to 0, we handle it in testbench.



This is for corner case for x=0 to x=255(after first 15 clock finish), it is same as above, so we don't go into details.

Originally for 1 line, it needs 81 * 256 * 256 input pixels, but with input buffering, it only needs (15+9*256)*256, which is nearly 1/9 of original.

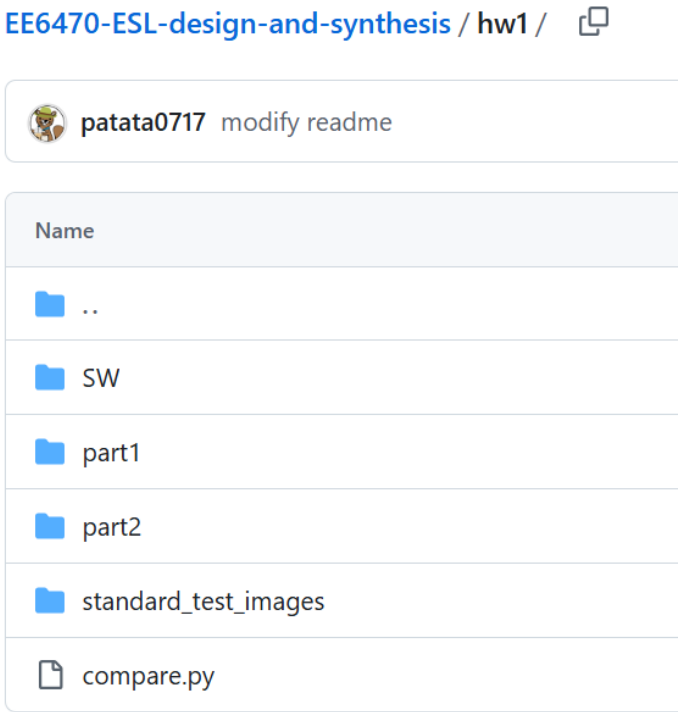
We know that, each filter had 10ns delay. Testbench need to print 9 data, testbench has 1ns delay, Gaussian filter need to print 3 data, Gaussian has 10ns delay, Laplacian need to print 1 data, Laplacian has 10ns delay. So the critical path is Gaussian, which has 30ns delay for 1 LoG output.

But, with FIFO channel, it can avoid the delay for critical path.

Input	Simulated time(ns)	Ratio
Lena 256, w/o buffer	5308416	100%
Mandril 512, w/o buffer	21233664	100%
Lena 256, w/i buffer	593664	11.18%
Mandril 512, w/i buffer	2366976	11.14%

So we did achieve nearly 1/9(11.11%) of original time.

6. File structure



In SW, it contains python to generate input image, as well as golden. And Cmake in part1, part2, will connect them as input and golden, and do comparison. And for parameter, for changing an new image input, we may need to change the parameters, I make it can be done inside Cmake, so the Cmake will generate adequate filter_def.h for the parameters of the system.

7. Discussion and difficulties

For part 1, we modify sobel filter code to implement the LoG filter, we need to change the header file of .bmp, then adjust channel structure.

For part 2, we use data buffer to reduce input pixel count, and achieve nearly 1/9 of original time cost.

One difficulty I faced is I can't implement the python rounding mechanism, I try round() function in C++ and other ways, but find it different from golden. So I modify the python code, to fix rounding method to (val + 8)/16 (Gaussian filter). And the systemC code is as above.

Python:

```
# Convolve and Save Output
image_gray_G2 = conv2d(image_gray, Gkernel, stride=1, padding=1)
image_gray_G2_scale = (image_gray_G2 + 8) // 16
cv2.imwrite(f'golden/{filename}_gray_noise_gaussian2.bmp', image_gray_G2_scale)
```

SystemC:

```
int gaussian_result = (val + 8) / 16;
```

We don't know how python handling rounding for Gaussian filter, which will divided by 16, so we forcefully make the python code do the integer rounding, which is (val + 8) / 16, up-rounding when 0.5. Otherwise, the rounding machenism is very difficult to achieve in hardware.