Lab Assignment 4

Topic: Reimplement into Convolutions

Assigned: **10/18/2019**
Due: **10/25/2019 noon**

This assignment continues the modeling of our application example, the Canny Edge Detector, as a proper system-level specification model which we can then use to design our SoC target implementation. We will then use this model for initial performance estimation in order to identify the components with the highest computational complexity.

1 Setup

Again, we will use the same setup as for the previous assignments. Start by creating a new working directory, so that you can properly submit your deliverables in the end.

```
mkdir lab4
cd lab4
```

2 Recode the Specification into Convolution-based Processing

The Canny edge detection source code we have started with uses many optimizations in order to reduce the amount of computations. While this leads to a more efficient implementation (in SW), it does not allow exploiting component reuse across the individual kernel operations. Each operation (Gaussian blur, derivative x, derivative y) in the un-optimized form are convolutions (`https://en.wikipedia.org/wiki/Kernel_(image_processing)`). This section of the lab has two objectives:

1. Convert to using general convolutions, which enables reusing the convolution kernel multiple times in the example.

2. Convert to a finer granularity of image lines to explore parallelism within processing one image.

2.1 Convert to Row Granularity and add Timing

Start with the template *lab4-skel.tar.gz*. It is based on the solution from lab 3, however has everything removed below the DUT.

The DUT has a pipelined hierarchy. To allow for this, two behaviors were added *QueueRX* and *QueueTx* they receive/send from/to queue. This was previously integrated in other behaviors (e.g. Gaussian_Smooth).

Note that the template also introduces new behaviors *Stimulus_Count* and *Monitor_Count* with insert known pixel values (first hex digit is the row, second hex digit is the column, image size 10x5) and print received values in the Monitor. The counting stimulus/monitor can be enabled with the *#define COUNT_STIM* in *defs.sh*.

Compile and run the specification. Validate that you are getting the same output image as the input image. Next, we need to add timing information.

1. Add timing emulation to the stimulus to simulate 10 fps at our resolution. Insert the appropriate *waitfor* statement. Use *#define* to define the timing constant. Document in your report (and in source code) how you have derived the timing. When the *Monitor* has received the last line, print the simulated time (which can be obtained by *now()*).

2. Compile and run your new specification. Validate that you are getting the same output image as the input image. Validate that the printed simulated time is 1/10 seconds.

Congratulations, now you are ready for the convolution fun.

2.2 Single Convolutions Combining both Computation and Buffering

To start with streaming convolutions, we focus on a kernel size of 3x3. Input image and output image should have the same dimensions. To simplify convolution processing, the convolution itself should not deal with border cases. Hence, a 0 padded input is expected. However, to make our lives easier, we will only 0 pad top and bottom (add new lines), but ignore the left and right border for now.

1. To process a 3x3 convolution 3 lines of pixels are needed. However, the DUT only receives one line at a time. First, we need to create buffers for *rowInL*, *rowInLL*. Create these buffers as piped types same as *rowIn*. Next, create new behavior *PipeBuffer*, it gets an input row and produces an output row of pixels. The main simply assigns the input to output. Instantiate the behavior in the DUT twice as *b1* producing *rowInL* and *b2* producing *rowInLL*. Insert the instances in the pipeline between *qRx* and *qTx*.

2. Compile and simulate the example. You will see a deadlock in the simulation. To solve the deadlock above, expand the Stimulus. After sending the actual rows of image data, keep sending blank rows. Reason in your lab report why the deadlock happens and why sending the blank rows fixes the deadlock – think about pipeline semantics.

3. Create a behavior Convolution, which takes 3 rows as an input, weights (type *tWeights*) and outputs a new tRowPixel. Start out with the simplest implementation: copy the middle row as an output row. Instantiate the Convolution in DUT and place after *b2* before *qTx*. Compile and simulate. The simulation should pass, however the outputs don't match. To debug the problem, switch to counting Stimulus / Monitor by defining *COUNT_STIM* in *defs.sh*. Hint: the problem is symmetric to the previous deadlock. In your report reason about how you solve this and why. Propagate the fix to both *Monitor_Count* and *Monitor*. After you have fixed this, you are ready to implement the convolution.

4. In the Convolution behavior create one loop over the number of columns, adjust the loop such that the first and last pixel in a row are not computed (border pixels). The loop iterator indicates the pixel column to be computed in the output row. In the loop body compute the cumulative sum with the neighboring pixels after multiplying with the weight from the kernel. Use floating point computation to get started with (weights are also in float), simply cast to tPixel in the end (we will improve in a later step).

5. Validate the functionality of the convolution with the identity kernel {{0,0,0},{0,1,0},{0,0,0}}. Include in your lab report the output for the count stimulus.

6. Define a kernel to shift the image one row bottom and to the right. Include in your lab report kernel and the output for the count stimulus.

7. Define a kernel to shift the image one row up and to the left. Include in your lab report kernel and the output for the count stimulus.

Congratulations! You made your first convolution from scratch in a pipelined fashion. However, to do some useful computation, the output result needs to be saturated according to the output data format. We will do this in the next step finding x derivatives as an example.

2.3 Simple Image Filter

To test the convolution, use a Sobel Operator (`https://en.wikipedia.org/wiki/Sobel_operator`). {{+1,0,-1},{+2,0,-2},{+1,0,-1}}.

1. Update the convolution weight definition according to the G_x Sobel operator.
2. Compile and run the example. The result will be distorted. Reason in your lab report why it is distorted.
3. Next, we improve the conversion from floating point in the convolution to integer in the output image. Simple casting a float to integer with truncate the result. To emulate rounding first add 0.5 to the cumulative pixel sum before casting.
4. Add code to observe the range boundaries (0 .. 255 for unsigned char).
5. Compile and simulate your results. Binary compare your results to 'beachbus-dx.pgm'.

Complete this step by validating that your refined model still compiles, simulates, and generates the correct output images. Ensure also that your code still compiles cleanly without any errors or warnings.

3 Write-up Instructions and Lab Assignment

1. Create a brief lab report
2. Answer each question explicitly made in the steps above. For easier grading, refer to where the question was made (e.g. 2.2.1 about the timing specification).
3. Briefly describe whether or not your efforts were successful and what (if any) problems you encountered. We will take this input into account when grading your submission. Please be brief!

Please submit your writeup as a PDF and your modified source code as a tar.gz archive. Your code should compile on the COE Linux systems. Validate the functional correctness of your code by comparing the compressed image against the reference code. Provide a Makefile to compile your modified code. Before creating the archive, please clean the build to remove any binary files. Submit the lab report as PDF (1) and the modified code (2) as a tar.gz file through Canvas. Please follow the naming convention with this format:

<myneu_id>_<hw|lab>_<Number>.<file_suffix>