EECS 222: Embedded System Modeling
Winter 2020

Assignment 8

Posted: February 18, 2020
Due: February 26, 2020 at 6pm

Topic: Pipelining and parallelization of the Canny Edge Decoder

1. Setup:

This assignment continues the modeling of our application example, the Canny Edge Detector. This time we will refine our model with back-annotated timing and pipeline and parallelize the components in the design-under-test (DUT) block. Over the course of the 6 steps outlined below, our design model will be refined from an untimed model into one with estimated delays where the simulation allows us to observe the improved performance due to pipelining and parallelization.

Again, we will use the same setup as for the previous assignments. Start by creating a new working directory with a link to the video files.

```
mkdir hw8
cd hw8
ln –s ~eecs222/public/video video
```

As in the previous assignments, you have again the choice of using either SpecC or SystemC for your modeling and estimation. Both SLDLs are suitable for this assignment, but several steps are different, as outlined in detail below.

As starting point, we will use a reference solution for Assignment 6. Note that in addition to the changes described in the shortened Assignment 6, this model also contains four children modules which have been separately instantiated in the **`Gaussian_Smooth`** module, namely **`Receive_Image`**, **`Gaussian_Kernel`**, **`BlurX`**, and **`BlurY`**.

The instance tree of the **`DUT`** module looks like this:

```
DUT canny
|------ Gaussian_Smooth gaussian_smooth
|       |------ Receive_Image receive
|       |------ Gaussian_Kernel gauss
|       |------ BlurX blurX
|       \------ BlurY blurY
|------ Derivative_X_Y derivative_x_y
```

```
|------ Magnitude_X_Y magnitude_x_y
|------ Non_Max_Supp non_max_supp
\------ Apply_Hysteresis apply_hysteresis
```

You can copy the prepared model as follows:

```
cp ~eecs222/public/cannyA6_ref.sc canny.sc
cp ~eecs222/public/cannyA6_ref.cpp canny.cpp
```

Note that the models for SystemC and SpecC differ here in their behavioral composition. The SpecC model uses sequential composition and port-mapped variables, while the SystemC model uses concurrent composition and port-mapped **`sc_fifo`** channels (SystemC).

You may also want to reuse the **`Makefile`** from the previous assignments:

```
cp ~eecs222/public/MakefileA5SpecC ./
cp ~eecs222/public/MakefileA5SystemC ./
```

As before, depending on whether you choose SpecC or SystemC, rename the corresponding file into the actual **`Makefile`** to be used by **`make`**.

2. Pipelining and Parallelization of the Canny Model

In order to observe the performance of the application in the simulator, we need to insert statements to monitor the simulated time in the test bench (Step 1) and then instrument the model with estimated delays in the DUT (Step 2).

Step 1: Instrument the model with logging of simulated time and frame delay

When we are interested in the latency and frame delay, we need to measure the time it takes to process a frame. To do that, we let the Stimulus block note the start time of processing each frame and communicate that to the Monitor which, in turn, can then compute and display the delay for each frame.

For the communication from Stimulus to Monitor, instantiate a FIFO channel with sufficient (!) buffer space for the frame start times. The channel (of type **`sc_time_queue`** in SpecC, or **`sc_fifo<sc_time>`** in SystemC, respectively) should pass time stamps from the Stimulus to the Monitor. In the Stimulus, take the current simulated time right after sending out the frame image, print it to the screen for observation, and also send it to the Monitor through the new channel. In the Monitor, take the difference between the current simulated time and the time the frame was sent, and display it on the screen for each frame.

The following log illustrates the expected screen output after this step (your results may vary in ordering):

```
0: Stimulus sent frame  1.
0: Stimulus sent frame  2.
0: Monitor received frame  1 with     0 ms delay.
0: Stimulus sent frame  3.
0: Monitor received frame  2 with     0 ms delay.
0: Stimulus sent frame  4.
0: Monitor received frame  3 with     0 ms delay.
[...]
0: Stimulus sent frame 20.
0: Monitor received frame 19 with     0 ms delay.
0: Monitor received frame 20 with     0 ms delay.
0: Monitor exits simulation.
```

As shown above, it is recommended to prefix each log line with the current simulated time as this significantly simplifies understanding and any needed debugging. Also shown above is the choice of milli-seconds (noted as **ms**) as the default time unit which fits well for our application.

You will want to keep a copy of your model at this stage, say **cannyA8_step1**, so that you can compare the observed timing among the different models in this assignment at the end.

Step 2: Back-annotate estimated timing in the DUT components

In the previous Assignment 7, we obtained some rough timing estimates for the blocks in the DUT. However, those measurements were performed on the server and do not reflect the performance of an embedded platform. So, we will use different measurements here instead.

In particular, we will use from now on timing estimates obtained for the Canny application on a Raspberry Pi 3 prototyping platform with a quad-core ARM-based processor. This timing has been measured using the same approach as step 2 in Assignment 7, except is was performed on a prototyping board.

Specifically, we will assume the following delays for the DUT components:

```
Receive_Image          0 ms
Make_Kernel            0 ms
BlurX               1880 ms
BlurY               2010 ms
Derivative_X_Y       530 ms
Magnitude_X_Y        910 ms
Non_Max_Supp         960 ms
Apply_Hysteresis     740 ms
```

Back-annotate these delays into your model by inserting suitable wait-for-time statements at the beginning of the main method of each DUT component.

After inserting the wait-for-time statements, run your model and observe the simulated time and frame delays reported by the log. Again, you want to keep a copy of your model at this stage, say **`cannyA8_step2`**, so that you can compare this observed timing with the following improved models.

Step 3: Improve the test bench to also log the frame throughput

As discussed in the lectures, the frame delay measured in Step 1 is helpful, but we are mostly interested in observing the performance of our model by means of its *throughput*, i.e. the *frames per second (FPS)* coming out of the video processing pipeline. To measure this, we will now extend the timing log produced by the test bench in Step 1.

Specifically, we let the Monitor module measure and report the frame throughput upon receiving a new frame. The extended log should look similar to the following at the end (however, depending on your specific model, your observed values may vary):

```
[...]
133570: Monitor received frame 19 with    28120 ms delay.
133570:   7.030 seconds after previous frame,  0.142 FPS.
140600: Monitor received frame 20 with    28120 ms delay.
140600:   7.030 seconds after previous frame,  0.142 FPS.
140600: Monitor exits simulation.
```

The frame throughput is observed in the Monitor module by measuring the arrival time of two consecutive frames and calculating the difference of the two timestamps. Converted to seconds, the reciprocal value is the desired FPS result.

Adjust your model to print the extra log line for each received frame. Again, keep a copy of the model at this stage, say **`cannyA8_step3`**, so that you can compare the observed timing with the following improvements.

Step 4: Pipeline the DUT into stages for each component

As discussed in the lectures, we will use pipelining as the overall technique to improve the throughput of the DUT.

If you are using SpecC for your modeling and communicate via regular variables, pipelining can be applied by simply replacing the endless loop in the **`Canny`** behavior (i.e. the **`fsm`** construct) with an infinite pipeline (i.e. a **`pipe`** construct). Then, to allow for the necessary buffering of the data between the pipeline stages, add **`piped`** qualifiers to any port-mapped variables between the stages. Note that you will need to duplicate those variables (and ports) whose values are needed in multiple following stages.

If you are using SpecC and your DUT components communicate via queue channels, then no **piped** qualifiers are needed. You can create pipeline behavior either by using a **pipe** construct, or by a **par** construct with endless loops around the stages.

If you are using SystemC and your DUT components are already communicating via **sc_fifo** channels, then there is nothing to do in this step. Your model is already pipelined!

As a result of this step, your model should contain 5 pipeline stages and, because of this, execute significantly faster (in simulated time!) than before.

Again, you want to keep a copy of your model at this stage, say **cannyA8_step4**.

Step 5: Integrate the Gaussian Smooth components into the pipeline stages

To further improve the performance of your design, we will also decompose the first pipeline stage, namely the Gaussian Smooth block, and create two additional pipeline stages for the **BlurX** and **BlurY** blocks. In other words, we move the **BlurX** and **BlurY** blocks from the **Gaussian_Smooth** parent one level up into the DUT. Here, be sure to properly arrange the port connectivity and add any needed buffering between the new pipeline stages.

The expected instance tree of the **DUT** block should look like this:

```
DUT
|------ Gaussian_Smooth gaussian_smooth
|       |------ Receive_Image receive
|       \------ Gaussian_Kernel gauss
|------ BlurX blurX
|------ BlurY blurY
|------ Derivative_X_Y derivative_x_y
|------ Magnitude_X_Y magnitude_x_y
|------ Non_Max_Supp non_max_supp
\------ Apply_Hysteresis apply_hysteresis
```

As a result of this step, your model should now contain a total of 7 pipeline stages and, once again, execute significantly faster (in simulated time) than before. (HINT: For a SystemC model, you might not see any improvement in the throughput, because the two pipeline stages in the Gaussian Smooth block already behaved as part of the overall pipeline in the previous step.)

As discussed in the lectures, the throughput for a model with overall pipeline structure is determined by the longest stage delay. So at this point, your throughput should depend on the **BlurY** block because that incurs the longest delay.

Again, keep a copy of your model at this stage, say **cannyA8_step5**.

Step 6: Slice the **BlurX** and **BlurY** blocks into parallel components

Finally we will remedy the performance bottleneck in the **BlurX** and **BlurY** components by parallelization. As discussed in the lectures, both blocks are straightforward to optimize by parallelizing the operations in the rows and columns, respectively. While we could technically operate on every single row or column in parallel (as a real graphics processing unit (GPU) would do it), we will limit our efforts to 8 parallel slices for this assignment.

Specifically, convert the existing **BlurX** and **BlurY** blocks into **BlurX_Slice** and **BlurY_Slice** components that only operate on a one-eighth slice of the image. For example, the first **BlurX_Slice** instance **sliceX1** will process the rows from **(ROWS/8)*0** through **(ROWS/8)*1-1** and **sliceX2** will process the rows from **(ROWS/8)*1** through **(ROWS/8)*2-1** and so on. Be sure to adjust the back-annotated delays by the expected speedup of 8x.

Then, instantiate 8 parallel instances of these slice processors in replacements of the previous **BlurX** and **BlurY** blocks. In the end, the expected hierarchy of the **DUT** should look like this:

```
DUT
|------ Gaussian_Smooth gaussian_smooth
|       |------ Receive_Image receive
|       \------ Gaussian_Kernel gauss
|------ BlurX blurX
|       |------ BlurX_Slice sliceX1
|       |------ BlurX_Slice sliceX2
|       |------ BlurX_Slice sliceX3
|       |------ BlurX_Slice sliceX4
|       |------ BlurX_Slice sliceX5
|       |------ BlurX_Slice sliceX6
|       |------ BlurX_Slice sliceX7
|       \------ BlurX_Slice sliceX8
|------ BlurY blurY
|       |------ BlurY_Slice sliceY1
|       |------ BlurY_Slice sliceY2
|       |------ BlurY_Slice sliceY3
|       |       [...]
|       |------ BlurY_Slice sliceY7
|       \------ BlurY_Slice sliceY8
|------ Derivative_X_Y derivative_x_y
|------ Magnitude_X_Y magnitude_x_y
|------ Non_Max_Supp non_max_supp
\------ Apply_Hysteresis apply_hysteresis
```

Note that we will use a simplified method of communication between the slices. Instead of queues or FIFO channels, we can use the image arrays in **`BlurX`** and **`BlurY`** as shared member variables among the slices.

HINT for SpecC: Rename the existing behavior **`BlurX`** into **`BlurX_Slice`** and only add two variable input ports, say **`rowStart`** and **`rowEnd`**. Then create a new behavior **`BlurX`** (with the same ports as the original **`BlurX`**) and instantiate 8 instances of **`BlurX_Slice`** in it where their ports are connected straight to the parent **`BlurX`** and the new ports **`rowStart`** and **`rowEnd`** are initialized with the bounds of their assigned slice. A simple **`par{ }`** construct over the 8 slice instances will then start them in parallel, wait for their completion, and complete the parent execution.

HINT for SystemC: There is no need to actually create an additional module type. Simply create 8 parallel **`SC_THREAD`**s inside the **`BlurX`** module. Each of those slice threads will need its own method that performs the blurring the same way as before, except only for its assigned slice from **`rowStart`** to **`rowEnd`**. Then, to ensure the parallel starting and ending of those slice threads, the main thread of **`BlurX`** reads the incoming frame data, notifies the slice threads via a **`start`** event, and waits for all of them to finish their slice calculation. To wait for all the 8 slice threads, the main thread needs to **`wait`** for an AND-combination of 8 events that are notified by the slice threads.

Apply these hints above also for the **`BlurY`** behavior or module, respectively.

As a result of this assignment, your final model **`cannyA8_step6`** should, once again, execute significantly faster (in simulated time) than in the previous step.

Note the timing of each model and report it in your text file submission. Specifically, we are interested in the total simulated time and the longest delay for processing a frame for each of the 6 steps of model refinement.

Thus, report the observed timings in the following table:

Model	Frame Delay	Throughput	Total time
CannyA8_step1	... ms		... ms
CannyA8_step2	... ms		... ms
CannyA8_step3	... ms	... FPS	... ms
CannyA8_step4	... ms	... FPS	... ms
CannyA8_step5	... ms	... FPS	... ms
CannyA8_step6	... ms	... FPS	... ms

3. Submission:

For this assignment, submit the following deliverables:

`canny.sc` *or* **`canny.cpp`**
`canny.txt`

As before, the text file should briefly mention whether or not your efforts were successful and what (if any) problems you encountered. In addition, include the observed timing results in the above table and a brief explanation.

To submit these files, change into the parent directory of your **`hw8`** directory and run the **`~eecs222/bin/turnin.sh`** script. As before, note that the submission script will ask for both the SystemC and SpecC models, but you need to submit only the one that you have chosen for your modeling.

Again, be sure to submit on time. Late submissions will not be considered!

To double-check that your submitted files have been received, you can run the **`~eecs222/bin/listfiles.py`** script.

For any technical questions, please use the course message board.

--
Rainer Dömer (EH3217, x4-9007, doemer@uci.edu)