EECS 222: Embedded System Modeling
Spring 2017

Assignment 7

Posted: May 24, 2017
Due: May 31, 2017 at 12pm (noon)

Topic: Pipelining and parallelization of the Canny Edge Decoder

1. Setup:

This assignment continues the modeling of our application example, the Canny Edge Detector, as a proper system-level specification model which we can use to design our SoC target implementation. This time we will refine the previous model with back-annotated timing and pipeline and parallelize the components in the design-under-test (DUT) block. Over the course of the 5 steps outlined below, the 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 hw7
cd hw7
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 certain task are different, as outlined in detail below.

As starting point, you can use your own SLDL model which you have created in the previous Assignment 6. Alternatively, you may start from the provided solution for Assignment 6 which you can copy as follows:

```
cp ~eecs222/public/CannyA6_ref.sc Canny.sc
cp ~eecs222/public/CannyA6_ref.cpp Canny.cpp
```

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

```
cp ~eecs222/public/MakefileA4SpecC ./
cp ~eecs222/public/MakefileA4SystemC ./
```

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

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

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

Since 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 of 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 simulation time stamps from the Stimulus to the Monitor. In the Stimulus, take the current simulation 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 time and the time the frame was sent, and display it on the screen for each frame.

The following log illustrates the desired screen output (with the exception that your log in this initial step will show all times as zero):

```
        0: Stimulus sent frame  1.
        0: Stimulus sent frame  2.
        0: Stimulus sent frame  3.
        0: Stimulus sent frame  4.
        0: Stimulus sent frame  5.
        0: Stimulus sent frame  6.
  6547500: Monitor received frame  1 with  6547500 us delay.
  6547500: Stimulus sent frame  7.
 13095000: Monitor received frame  1 with 13095000 us delay.
 13095000: Stimulus sent frame  8.
 19642500: Monitor received frame  2 with 19642500 us delay.
 19642500: Stimulus sent frame  9.
[...]
 91665000: Monitor received frame 13 with 39285000 us delay.
 91665000: Stimulus sent frame 20.
 98212500: Monitor received frame 14 with 39285000 us delay.
104760000: Monitor received frame 15 with 39285000 us delay.
111307500: Monitor received frame 16 with 39285000 us delay.
117855000: Monitor received frame 17 with 39285000 us delay.
124402500: Monitor received frame 18 with 39285000 us delay.
130950000: Monitor received frame 19 with 39285000 us delay.
130950000: Monitor exits simulation.
```

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

You want to keep a copy of your model at this stage, say **`CannyA7_step1`**, so that you can compare the observed timing among the different models in this assignment at the end.

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

At the end of the previous Assignment 6, we obtained some relative timing estimates for the blocks in the DUT. While these estimates are very rough and inaccurate at this stage, they can still serve the purpose of observing the benefits of the transformations we apply to the model in this assignment.

For consistency and easier discussion of this assignment, we will choose here the timing estimates obtained by SCE for the ARM_926EJS processor (see slide 11 of Lecture 15) assuming we can improve those by 10x for a 1.0 GHz clock frequency. Specifically, we will assume the following total delays for the DUT components:

```
Receive_Image       0 ms
Gaussian_Kernel     0 ms
BlurX               41680 ms
BlurY               43740 ms
Derivative_X_Y      5340 ms
Magnitude_X_Y       5340 ms
Non_Max_Supp        27200 ms
Apply_Hysteresis    7650 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. Here, be sure to adjust the above values for the number of frames processed in your test bench (the values above were estimated over the entire stream of 20 frames).

After inserting the wait-for-time statements, run your model and observe the simulation time and frame delays reported by the log. (Hint: if you model in SpecC, you should see the exact same times as listed in the example log above.)

Again, you want to keep a copy of your model at this stage, say **`CannyA7_step2`**, so that you can compare this initial observed timing with the following models in this assignment.

Step 3: Pipeline the DUT into stages for each component

As discussed in Lecture 15, we will use pipelining as the overall technique to improve the throughput of the DUT.

If you are using SpecC for your modeling, 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 the 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 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 **`CannyA7_step3`**.

Step 4: Integrate the Gaussian Smooth components into the pipeline stages

To further improve the performance of your design, we will 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.

Again, keep a copy of your model at this stage, say **`CannyA7_step4`**.

Step 5: Slice the **`BlurX`** and **`BlurY`** blocks into parallel components

Finally we will remedy the identified performance bottleneck in the **`BlurX`** and **`BlurY`** components by parallelization. As discussed in Lecture 15, 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) could 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 instance tree 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
```

As a result of this assignment, your final model **CannyA7_step5** 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 simulation time and the longest delay for processing a frame for each of the 5 steps of model refinement. Thus, report the observed timings in the following table:

Model	Frame Delay	Total simulation time
CannyA7_step1	... us	... us
CannyA7_step2	... us	... us
CannyA7_step3	... us	... us
CannyA7_step4	... us	... us
CannyA7_step5	... us	... us

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 **hw7** 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 Doemer (EH3217, x4-9007, doemer@uci.edu)