EECS 222: Embedded System Modeling
Winter 2020

Assignment 9

Posted: February 25, 2020
Due: March 4, 2020 at 6pm

Topic: Throughput optimization of the Canny Edge Decoder

1. Setup:

This assignment is the final chapter in the modeling of our application example, the Canny Edge Detector, as a system-level specification model suitable for SoC implementation. Here, we will optimize the pipelined DUT model obtained in the previous assignment so that the pipeline stages are optimized, better balanced, and therefore the throughput of the design is improved.

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 hw9
cd hw9
ln –s ~eecs222/public/video video
```

As before, you have the choice of using either SpecC or SystemC for your modeling.

As starting point, you can use your own SLDL model which you have created in the previous Assignment 7 and Assignment 8. Alternatively, you may start from the provided solution files for Assignment 7 and Assignment 8 which you can copy as follows:

```
cp ~eecs222/public/cannyA7_ref.cc cannyA9_step1.cc
cp ~eecs222/public/cannyA8_ref.sc cannyA9_step2.sc
cp ~eecs222/public/cannyA8_ref.cpp cannyA9_step2.cpp
```

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

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

Again, depending on whether you design in SpecC or SystemC, rename the corresponding file into the actual **`Makefile`** to be used by **`make`**.

Finally, we will use again the **ImageDiff** tool in this assignment which you can access via a symbolic link:

```
ln -s ~eecs222/public/ImageDiff ImageDiff
```

We will use this tool for comparing the generated images instead of the previously used Linux **diff** tool, as outlined in the instructions below.

2. Throughput optimization of the Canny Edge Decoder model

Step 1: Obtain optimized timing delays by utilizing compiler optimizations

In Assignment 7 Step 2, we measured the application performance on the department server with the Linux **clock()** API. However, we did not take any compiler optimizations into account.

Compiler optimizations are an easy choice to improve software program performance since the GNU compiler offers many optimization options for generating faster executables. So we run the compiler anew with optimizations enabled, and then measure the timing again.

A general-purpose optimization flag for the GNU compiler is **-O2** which we will use here. Other possible options include **-O3** and many other specific optimization flags. While you are welcome to test those as well, we will use the general-purpose **-O2** option in this step.

So, compile your model with compiler optimizations turned on and execute it again. Then compare the measured timing against the measurements from Assignment 7 and calculate the obtained speedup in a table, as follows:

```
T1  =  ...ms / ...ms = ...
T2  =  ...ms / ...ms = ...
T3  =  ...ms / ...ms = ...
T4  =  ...ms / ...ms = ...
T5  =  ...ms / ...ms = ...
T6  =  ...ms / ...ms = ...
T7  =  ...ms / ...ms = ...
Tot =  ...ms / ...ms = ...
```

Submit the filled table in your text file **canny.txt**. Keep a copy of your model at this point and name it **cannyA9_step1**.

Step 2: Back-annotate the optimized timing for the DUT components

The compiler optimizations applied and measured in Step 1 affect the DUT components differently and result in varying speedups. For simplicity (and easier

grading) purposes, however, we will assume now that the speedup is 2.5 on average and is also achieved on the prototyping board.

Thus, adjust the timing delays which we back-annotated into your model in Assignment 8 Step 2 accordingly. Specifically, we assume here that each DUT component will incur a speedup of 2.5x due to compiler optimizations.

After adjusting the wait-for-time statements, run your model and observe the simulation time and frame delays reported by the log. Again, you want to keep a copy of your model at this stage, say **`cannyA9_step2`**, so that you can compare this observed timing with the other models.

Step 3: Replace the Raspberry Pi 3 measurements with Raspberry Pi 4

As an example of optimization by use of “better” hardware, we will now assume that we switch from the Raspberry Pi 3 prototyping board to the next generation, Raspberry Pi 4, which is significantly faster than before.

Specifically, we have measured the following delays for the DUT components on a new RPi 4 board (using the same approach as described in Assignment 7):

```
Receive_Image          0 ms
Make_Kernel            0 ms
BlurX                440 ms
BlurY                625 ms
Derivative_X_Y       260 ms
Magnitude_X_Y        170 ms
Non_Max_Supp         320 ms
Apply_Hysteresis     295 ms
```

Back-annotate these new delays into your model by replacing the previous delays for each DUT component. Note that can assume that the 2.5x speedup due to compiler optimizations (Step 2 above) still applies also for these new base measurements.

Again, you want to keep a copy of your model at this stage, say **`cannyA9_step3`**, so that you can compare this observed timing with the other models.

Step 4: Replace floating-point arithmetic with fixed-point calculations (NMS only)

In order to further improve the throughput of our video processing pipeline, we need to balance the load of the pipeline stages. Specifically, we need to improve the stage with the longest stage delay.

In the following, we will experiment with fixed-point arithmetic that can often improve execution speed when floating-point operations are too slow. In other

words, we want to replace existing floating-point calculations by faster and cheaper fixed-point arithmetic with an acceptable loss of accuracy.

In our model in particular, the **Non_Max_Supp** behavior/module is now the bottleneck in the pipeline that we want to speed-up. Also, the **Non_Max_Supp** module is a good target where we can easily apply fixed-point optimization. (Generally, this technique can be applied also to other components, but we will limit our efforts to only the **Non_Max_Supp** block in this assignment.)

Find the **non_max_supp** function in the source code of your model. Identify those variables and statements which use floating-point (i.e. **float** type) operations. There are only 4 variables defined with floating-point type. Change their type to integer (**int**).

Next, we need to adjust all calculations that involve these variables. In particular, we need to add appropriate shift-operations so that the integer variables can represent fixed-point values within appropriate ranges. Since the details of such arithmetic transformations are beyond the scope of this course, we provide specific instructions here.

Locate the following two lines of code:

```
xperp = -(gx = *gxptr)/((float)m00);
yperp = (gy = *gyptr)/((float)m00);
```

Comment out those lines and insert the following statements as replacement:

```
gx = *gxptr;
gy = *gyptr;
xperp = -(gx<<16)/m00;
yperp = (gy<<16)/m00;
```

To ensure functional correctness, compile and simulate your model. However, don't be disappointed if your **make test** fails! Note that the current **Makefile** compares the generated frames against the reference images and expects exact matches. Our arithmetic transformation, however, is not guaranteed to be exact. It is only an approximation!

In order to determine whether or not fixed-point arithmetic is acceptable for our application, we need to compare the image quality. You can do this by looking at them (e.g. use **eog** to display them on your screen), or better by using the provided **ImageDiff** tool. For example, use **ImageDiff** as follows:

```
./ImageDiff Frame.pgm video/Frame.pgm diff.pgm
```

You may want to adjust your **Makefile** so that the previously used Linux **diff** command is replaced with the **ImageDiff** tool instead.

Decide for yourself whether or not you find the changes incurred due to the use of fixed-point arithmetic acceptable for our edge detection application.

Generally, the cost of inaccurate images comes with the benefit of improved execution speed. Here, we will assume that the NMS block, which took 320ms to compute (before compiler optimizations), now only takes 290ms (without compiler optimizations). Apply this improved timing to your model (and assume the same 2.5x speedup due to compiler optimizations as in Step 2). Recompile and execute the model again to observe the improved performance in the simulation log.

Decide for yourself whether or not this change is worth it for our real-time video goal and explain your choice in the **`canny.txt`** text file. Also, report the observed timings in your **`canny.txt`** file in the following table:

Model	Frame Delay	Throughput	Total time
cannyA9_step2	... ms	... FPS	... ms
cannyA9_step3	... ms	... FPS	... ms
cannyA9_step4	... ms	... FPS	... ms

3. Submission:

For this assignment, submit the following deliverables:

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

To submit these files, change into the parent directory of your **`hw9`** 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)