Embedded Systems Modeling and Design
ECPS 203
Fall 2017

Assignment 9

Posted: November 29, 2017
Due: December 6, 2017 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 an embedded system model in SystemC suitable for SoC implementation. Here, we will optimize the pipelined DUT model obtained in the previous assignment so that the pipeline stages are better balanced and therefore the throughput of the design is improved. We also apply further optimizations to reduce the execution time of the pipeline stages.

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 ~ecps203/public/video video
```

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

```
cp ~ecps203/public/CannyA8_ref.cpp Canny.cpp
```

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

```
cp ~ecps203/public/MakefileA5 Makefile
```

As before, you will need to adjust your stack size again. If you use the **`csh`** or **`tcsh`** shell, use:

```
limit stacksize 128 megabytes
```

On the other hand, if you use the **`sh`** or **`bash`** shell, use this:

```
ulimit -s 128000
```

Finally, we will use one more tool named **`ImageDiff`** in this assignment which you can copy as well:

```
cp ~ecps203/public/ImageDiff ./
```

We will use this **`ImageDiff`** 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: Improve the test bench to include the logging of frame throughput

As we will discuss in Lecture 19, 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 extend the timing logs produced by the test bench.

Specifically, we let the Monitor module measure and report the frame throughput upon receiving a new frame. A sample output fragment shall look like this:

```
[...]
17282 ms: Monitor received frame 14 with 14720 ms delay.
17282 ms:   1.030 seconds after previous frame,  0.971 FPS.
17842 ms: Stimulus sent frame 30.
18312 ms: Monitor received frame 15 with 15323 ms delay.
18312 ms:   1.030 seconds after previous frame,  0.971 FPS.
[...]
```

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 time stamps. Converted to seconds, the reciprocal value is the desired FPS result.

Adjust your model to print the extra log line for each received frame. Keep a copy of the model at this stage, say **`CannyA9_step1`**, so that you can compare the observed timing with the following improvements.

Hint: The throughput timing shown during simulation should match the longest stage delay in the video pipeline. For example, the above throughput delay of 1.03 seconds matches the back-annotated timing of the **`Magnitude_X_Y`** module in the previous Assignment 8 because that is the slowest pipeline stage in that model.

Step 2: Reduce the timing delays of the stages in the pipeline

As discussed in Lecture 17, we can exploit further options for improving the performance of our Canny Edge Detector implementation, beyond the pipelining and parallelization that we have already applied in Assignment 8. One easy option is to enable compiler optimizations.

Review Assignment 7 where we measured the timing of the major Canny functions on the Raspberry Pi prototyping board. Recall that we compiled the application without optimization:

```
g++ -Wall canny.c –o canny
```

Given that the GNU compiler offers many optimization options for generating faster executables, run the compiler again with optimizations enabled, and then measure the timing again.

A general-purpose optimization flag for the GNU compiler is **`-O2`** which you should test. Other possible options include **`-O3`**, **`-mfloat-abi=hard`**, **`-fmpu=neon-fp-armv8`**, and **`-mneon-for-64bits`** (see https://gist.github.com/fm4dd/c663217935dc17f0fc73c9c81b0aa845 for reference).

Experiment with several options and measure the timing of the Canny functions for each of them. Choose the best result and take note of these values. Then back-annotate the improved timing into the source code of your SystemC model and simulate it for correctness validation. The throughput should be visibly improved in a significantly higher FPS rate.

Keep a copy of your model with the updated timing at this point and name it **`CannyA9_step2`**.

Step 3: Replace floating-point arithmetic with fixed-point calculations (NMS module 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 optimize the stage with the longest stage delay. In the following, we will experiment with fixed-point arithmetic that often can improve execution speed when floating-point operations are slow. In other words, we want to replace existing floating-point calculations by faster and cheaper fixed-point arithmetic with an acceptable loss in accuracy.

For this step in particular, we will assume that the **`Non_Max_Supp`** module is a bottleneck in our pipeline that we want to speed-up. In the Canny algorithm, the **`Non_Max_Supp`** module is a good target where we can easily apply this 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);
```

Then, comment those lines out and insert the following replacement statements:

```
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 **`Makefile`** used so far 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 the images (e.g. use **eog** to display them on your screen), or better by using the provided **ImageDiff** tool. This command-line tool is built from Canny source code functions and compares the individual pixels of two input images (first and second argument) and generates an output image (third argument) which shows the differences. It also reports the number of mismatching pixels found. 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 by using 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. At the same time, measure the execution time of the modified **non_max_supp** function on your prototyping board (after applying the fixed-point modification to the source code) and decide whether or not this change is worth it for our real-time video goal.

Regardless what you decide, keep a copy of your model at this point and name it **CannyA9_step3**.

As discussed in Lecture 17, if the throughput of your model at this stage does not meet the real-time goal of 30 FPS yet, you can still decide to reduce some requirements for our application, for example, reduce the image size or simply accept a lower FPS rate for the end user. You can discuss this in detail in your final report for this course.

3. Submission:

For this final assignment, submit the following deliverables:

Canny.cpp	(your final SystemC model)
Canny.txt	(brief description with your throughput results)

To submit these files, change into the parent directory of your **hw9** directory and run the **~ecps203/bin/turnin.sh** script. To double-check that your submitted files have been received, you can run the **~ecps203/bin/listfiles.py** script.

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

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

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