Embedded Systems Modeling and Design
ECPS 203
Fall 2018

Assignment 9

Posted: November 28, 2018
Due: December 5, 2018 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 model obtained in the previous assignment so that the pipeline stages are better balanced and therefore the throughput of the design is improved. In particular, we apply more optimizations to reduce the execution time of the longer 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

For maximum throughput, a pipeline needs balanced minimum stage delays. So we will improve the balance by minimizing the longest stages.

Step 1: Turn on compiler optimizations for maximum execution speed

As discussed in the lectures, 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 choice is to enable compiler optimizations.

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

```
g++ -Wall canny.cpp -o canny
```

Given that the GNU compiler offers many optimization options for generating faster executables, 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 you should test. Other possible options include **`-O3`**, **`-mfloat-abi=hard`**, **`-fmpu=neon-fp-armv8`**, and **`-mneon-for-64bits`**, and others (for reference, see for instance https://gist.github.com/fm4dd/c663217935dc17f0fc73c9c81b0aa845).

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 validation of correctness. The throughput should be improved in a significantly higher FPS rate.

Keep a copy of your model with the updated timing at this point and name it **`CannyA9_step1`**.

Step 2: Consider fixed-point calculations instead of floating-point arithmetic (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 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 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);
```

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 **Makefile** used so far compares the generated frames against the reference images and expects exact matches. This 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, optional) 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 calling 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_step2**.

As discussed in the lectures, 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. Project Report

In lieu of a final exam in this course, we will use a technical report about the project. This project report shall cover all the steps taken in our design of the embedded system model of the Canny Edge Detector. That is, the report shall describe and document all the modeling tasks performed from the starting point of downloading the application source code (Assignment 1) down to creating the final model obtained in this Assignment 9, and provide the background and *reasoning* for taking the performed steps and design decisions.

Your final technical project report shall carry the title *"Modeling of a Canny Edge Detector for Embedded Systems Design"* with subtitle of the course number and course title. The contents of the report shall describe the overall "story" of our project, all the way from downloading the initial C reference code, via describing and simulating the application in SystemC language, to our modeling and optimization for real-time video. The report should focus on the reasoning behind the optimizations performed, and conclude with a summary of the lessons learned.

Specifically, the technical report should be structured as follows:

1. Title page
    - Project title, author, date, course number and title
    - Abstract
2. Introduction
    - Embedded system modeling and design concepts
    - The IEEE SystemC language
3. Case study of a Canny Edge Detector for Real-time Video
    - Structure of the Canny edge detection algorithm
    - Modeling and simulation in IEEE SystemC
    - Model refinement for pipelining and parallelization
    - Performance estimation and throughput optimization
    - Real-time video performance results
4. Summary and Conclusion
    - Lessons Learned
    - Future work
5. References

As a guideline, your project report should be about 12 pages in length, including the title page, all figures and tables, and appropriate references.

4. Submission:

For this final assignment, submit the following deliverables:

`Canny.cpp` (your final SystemC model, graded)
`Canny.txt` (extended performance table from A8 with new results from A9, graded)
`Canny.pdf` (draft project report, to be reviewed but not graded)

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)