Specification and Modeling of a Canny Edge Detector for System-on-Chip Design

EECS 222 - Embedded System Modeling

Salam Zantout

June 14, 2017

Abstract

Embedded system modeling was performed on a specific image processing application, an implementation of the Canny Edge Detector algorithm. The application was first converted from single image processing to real-time video handling (processing a sequence of images extracted from a stream of video frames). After that, a suitable embedded system model of this application was designed and described in a System-Level Description Language (SLDL), SpecC. Finally, the specification model was simulated for functional and timing validation, and then refined for synthesis and implementation as an embedded System-on-Chip (SoC) suitable for use in a digital camera.

I- Introduction

System-level modeling and design

System modeling is the interdisciplinary study of the use of models to conceptualize and construct systems [1]. The top down system design flow starts first with the product features, then the specification model, followed by the architecture model. Next comes the communication model, then the implementation model and finally manufacturing [2]. There are different models of computation to formally and abstractly describe a system: State-based Models (FSM, DFG, FSMD, SFSMD, HCFSM, PSM), Process-based Models (Processes and threads, Kahn Process Network, Synchronous Data Flow) and Imperative Programming Models (C/C++) [2].To design an embedded system, a design flow and methodology are needed. The design space should be explored, hardware and software should be co-designed, embedded architecture and network should be explored and synthesized. System software/hardware interface should be generated while taking into consideration the real-time constraints, specification-to-architecture mapping, design tools and methodologies [3].

Essential concepts and coverage in SpecC/SystemC SLDL

The goals and requirements of a System-Level Description Languages (SLDL) are Formality (Formal syntax and semantics), Executability (Validation through simulation), Synthesizability (Implementation in Hardware and/or Software and Support for IP reuse), Modularity (Hierarchical composition and Separation of concepts where computation is encapsulated in modules / behaviors and communication is encapsulated in channels), Completeness (Support for all concepts found in embedded systems), Orthogonality (Orthogonal constructs for orthogonal concepts) and Simplicity (Minimality) [2].
Figure 1 below summarizes the coverage of some of these concepts in SpecC and SystemC SLDL.

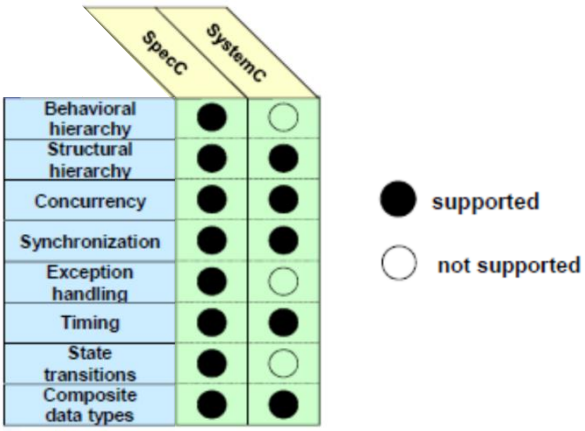

Figure 1: Essential Concepts in SpecC and SystemC [2]

II- Case study using the Canny Edge Detector application

For the embedded system modeling project in this course, a specific image processing application was chosen, namely an implementation of the Canny Edge Detector algorithm [4]. The overall project goal was to design a suitable embedded system model of this application and describe it in a System-Level Description Language (SLDL). This embedded specification model was not only simulated for functional and timing validation, but was also then refined for synthesis and implementation as an embedded System-on-Chip (SoC) suitable for use in a digital camera.

Obtaining and studying the Canny application

The first step was to download the algorithm and get familiar with its functionality. The Canny Edge Detector algorithm takes an input image, e.g. a digital photo, and calculates an output image that shows only the edges of the objects in the photo. The processing steps are shown in figure 2.

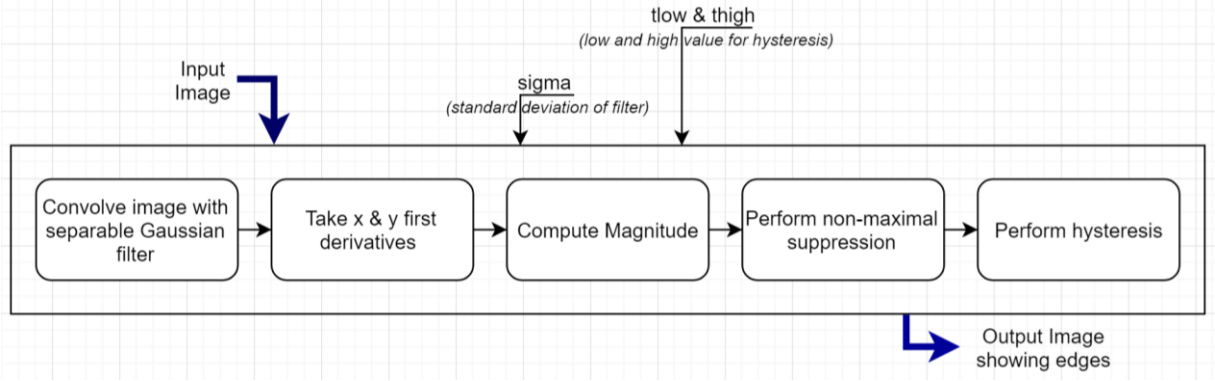

Figure 2: Canny Processing Steps

The C reference implementation of the Canny edge detection algorithm was used as the starting point for the embedded design model. A golf cart image served as an initial input image. One of the functions present in the C code is called "Canny", it is where the actual computation occurs, and it is the one with the most complexity. Since C code does not have time limits, the application can't be expected to run in real-time as required by our overall goal.

Creating a simulatable model in SpecC/SystemC SLDL

Next, a clean SLDL Model was created for Simulation. First, a bug in the *non_max_supp* function was fixed. Specifically, the bug was that the suppression of non-maximum points in the algorithm incorrectly omitted a column of pixels at the right and a row of pixels at the bottom of the image. Second, the source code was cleaned-up so that there are no compilation warnings. Third, the used dynamic memory allocation was removed since it is not feasible in hardware implementation. In fact, a SoC cannot instantiate a new memory chip at runtime and so instead, static arrays with fixed sizes at compile time were used. Finally, in order to synthesize the model into an actual hardware chip, the configuration parameters which were flexible in the initial software (sigma, thigh, tlow, rows, cols, infilename), became fixed constants for the SoC implementation.

Creating structural hierarchy with test bench

The next step was modeling the application example, the Canny Edge Detector, as a proper system-level specification model which can then be used to design the SoC target implementation. The application was first converted from single image processing to real-time video handling (processing a sequence of images extracted from a stream of video frames). Instead of having an input image, we have a loop of 20 images that together form the video.

Then a model with a suitable top-level structural hierarchy was created, including a test bench as shown in figure 3. The Main behavior instantiates the Stimulus, Platform and Monitor behaviors in parallel. The Stimulus block reads the 20 input images (using a for loop of size 20) from the file system and passes it into the Platform via the first queue channel (q1). Correspondingly, the Monitor receives via the second channel (q2) the 20 generated edge images (using a for loop of size 20) from the Platform and writes it out into the output file. In the Platform, the DataIn block is in an endless loop. When it receives an input image it passes it unmodified to the DUT. Similarly, the DataOut block is also in an endless loop. When it receives

an input image from the DUT, it then passes it on. Finally, the DUT block contains the entire Canny algorithm source code. Its main thread receives an image via the input port, calls the canny() function to process it, and then sends out the edge image via the output port. Since the target SoC will never stop working (unless its power is turned off), this processing runs in an endless loop, similar to the infinite loops in the DataIn and DataOut blocks.

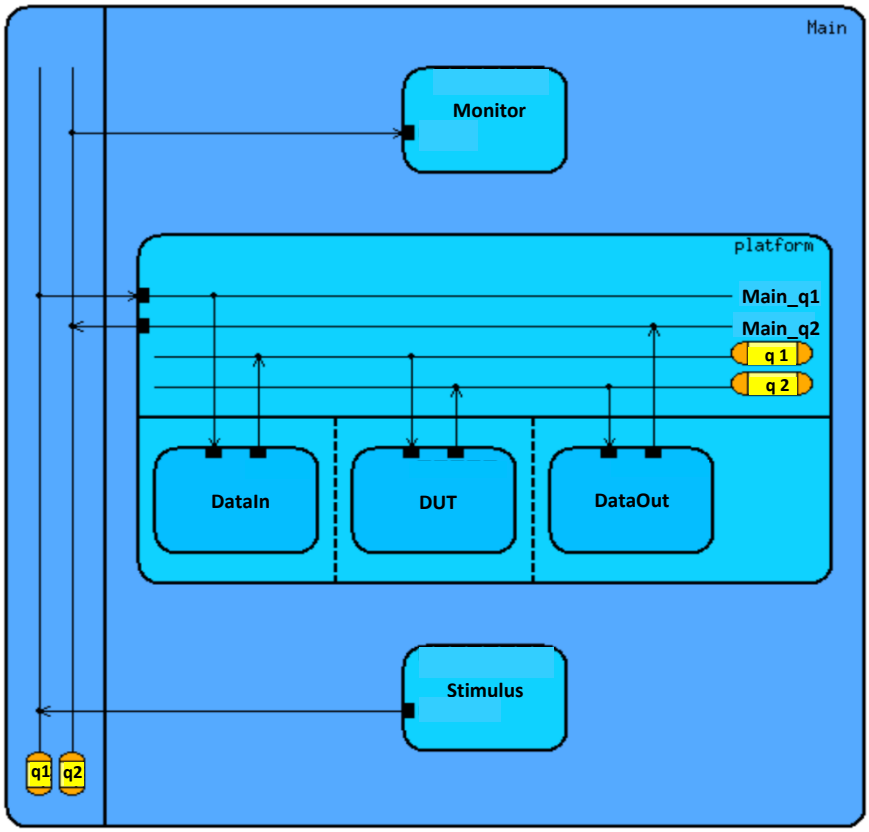

Figure 3: A5 Top-Level Structural Hierarchy with Test Bench Model

Pipelining and parallelization

Afterwards, the model was refined with a suitable structural hierarchy inside the design-under-test (DUT) block as shown in figure 4. The original canny function in the previous step consisted of a sequence of function calls to five Functions, namely gaussian_smooth, derivative_x_y, magnitude_x_y, non_max_supp, and apply_hysteresis. While in the previous model, these were all local methods in the DUT, in this step, they were wrapped into separate blocks (child behaviors) by themselves. The Canny behavior is now a sequential composition of its children. For communication, the child behaviors are connected by ports directly mapped to connecting variables (shown in green in the bottom right of figure 4). After that, the Gaussian Smooth component consisting of several steps was wrapped into separate children blocks, namely Receive_Image, Gaussian_Kernel, BlurX, and BlurY. As in the previous separation, this hierarchy level was equipped with the appropriate interconnections consisting of port-mapped variables (shown in green in the bottom left of figure 4).

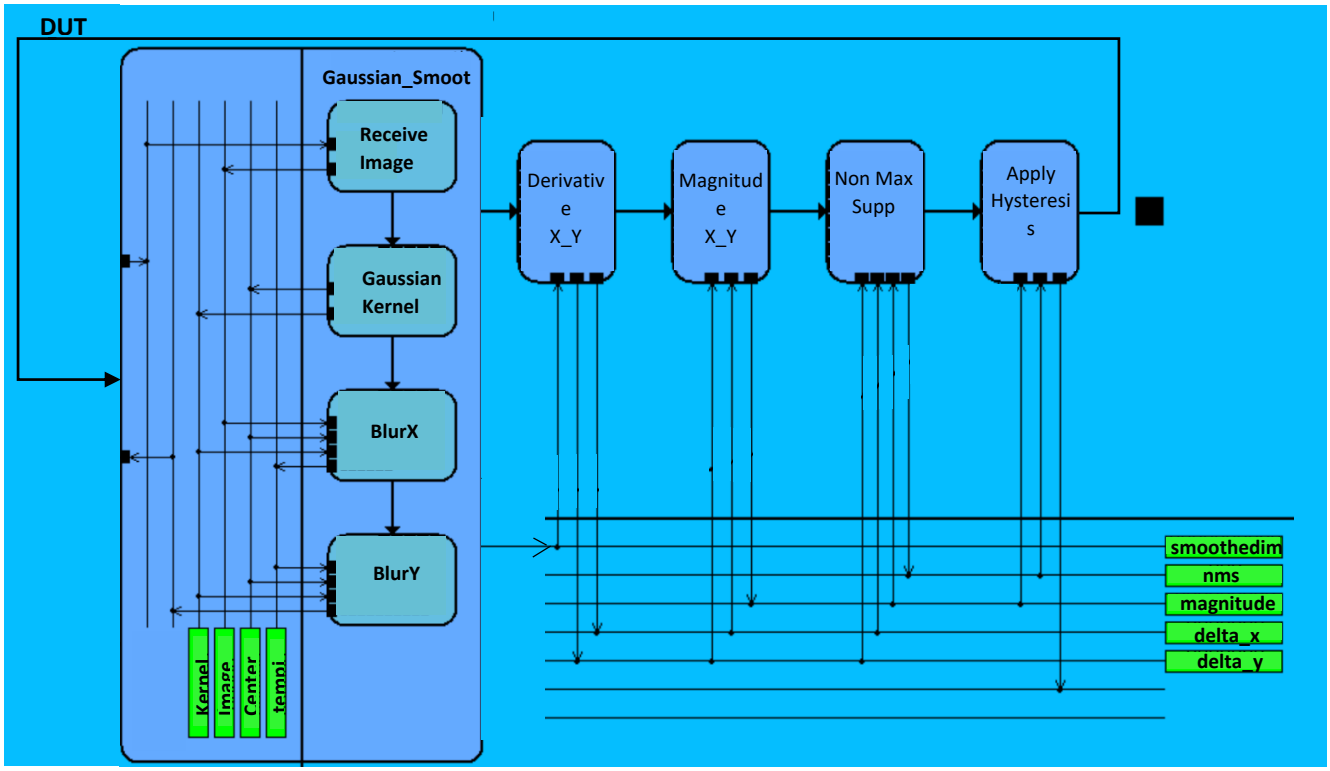

Figure 4: Structural Hierarchy inside the DUT

Finally, this model was used for initial performance estimation in order to identify the components with the highest computational complexity. The complexity comparison table for the blocks in the DUT is presented below in table 1.

Table 1: Complexity Comparaison Table

Gaussian_Smooth	58 %		
	Gaussian_Kernel 0.0 %	BlurX 49.9 %	BlurY 50.1 %
Derivative_X_Y	8 %		
Magnitude_X_Y	7 %		
Non_Max_Supp	17 %		
Apply_Hysteresis	10 %		
Total	**100%**		

These percentages were generated by examining the numerical form of the computation [operations] in the profiler integrated into the System-on-Chip Environment (SCE). As expected, The Gaussian_Smooth behavior has the highest computation percentage. More specifically, the BlurX and BlurY behaviors inside constitute the biggest computation behaviors with ~ 50% each which can also be seen in figure 5. The results in figure 5 can be explained by looking at the code of these 2 behaviors and noticing the 3 nested for loops that run a number of "windowsize" x "cols" x "rows".

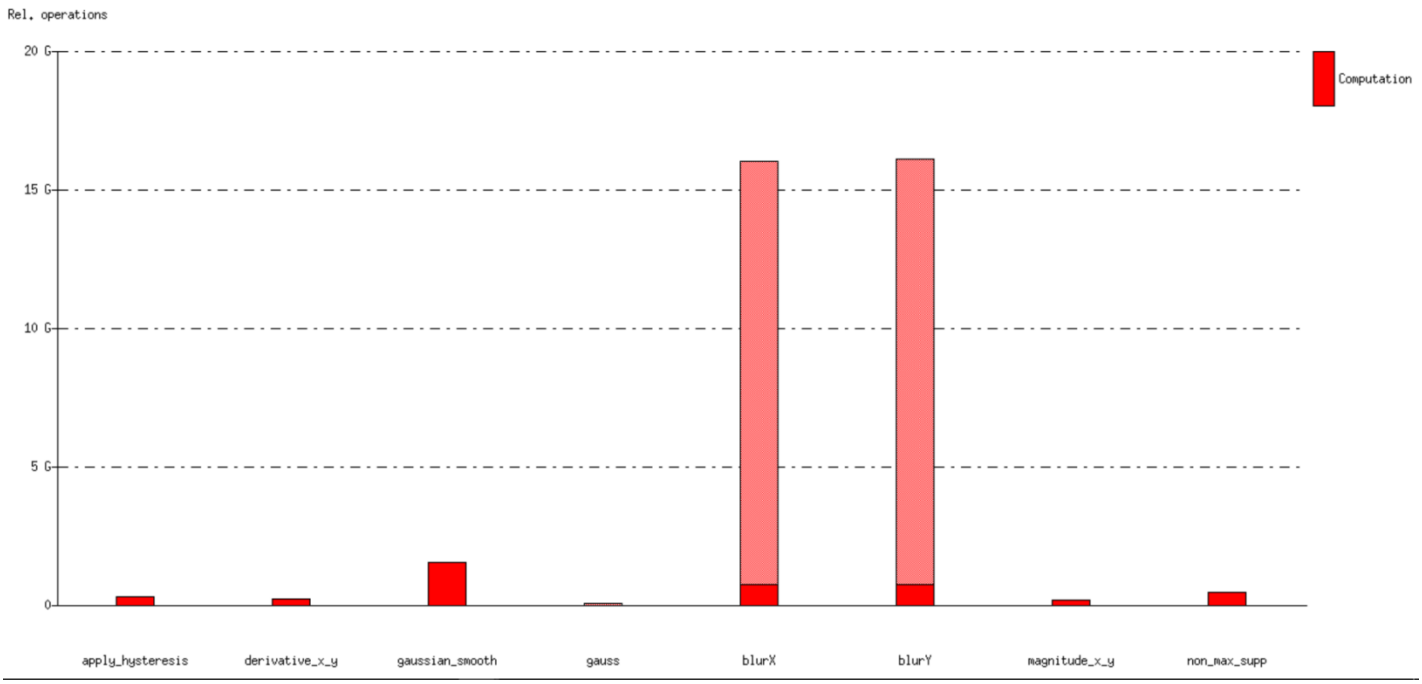

Figure 5: Computation Profile of Canny Blocks

Next, the previous model was refined to one with back-annotated timing and pipeline. Then the components in the design-under-test (DUT) block were parallelized. Generally, the design model was refined from an untimed model into one with estimated delays where the simulation allowed observing the improved performance due to pipelining and parallelization.

In order to observe the performance of the application in the simulator, statements to monitor the simulation time in the test bench were inserted. More specifically, the time it takes to process a frame was measured to determine the latency and frame delay. The Stimulus block noted the start time of processing each frame and communicated that to the Monitor through a channel. The monitor, in turn, then computed and displayed the delay of each frame. Specifically, the following delays were back-annotated for the seven stages in the DUT pipeline:

- Gaussian_Smooth → 0 ms
- BlurX → 2084 ms
- BlurY → 2187 ms
- Derivative_X_Y → 267 ms
- Magnitude_X_Y → 267 ms
- Non_Max_Supp → 1360 ms
- Apply_Hysteresis → 382.5 ms

These timing estimates were obtained by the SCE when using a single ARM_926EJS processor after assuming an improvement of the default clock frequency by a factor of 10 from 100 MHz to 1.0 GHz.

The frame delay and the total simulation time decreased significantly because of the pipelining used to improve the throughput of the DUT. Pipelining was applied by simply replacing the endless loop in the Canny behavior with an infinite pipeline. Then, to allow for the necessary buffering of the data between the pipeline stages, piped qualifiers were added to the port-mapped variables between the stages. Then the

stages of the pipeline were balanced by separating BlurX and BlurY and wrapping them into separate stages in the pipeline and hence got the diagram shown in figure 6.

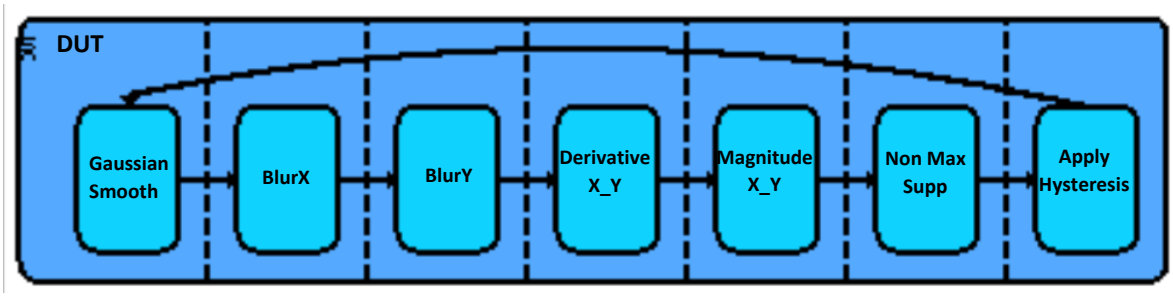

Figure 6: Pipelined DUT Behavior

Finally, as a second step of balancing, both behaviors (BlurX and BlurY) were parallelized by unloading the "for loops" 8 times and obtained the diagram shown in figure 7.

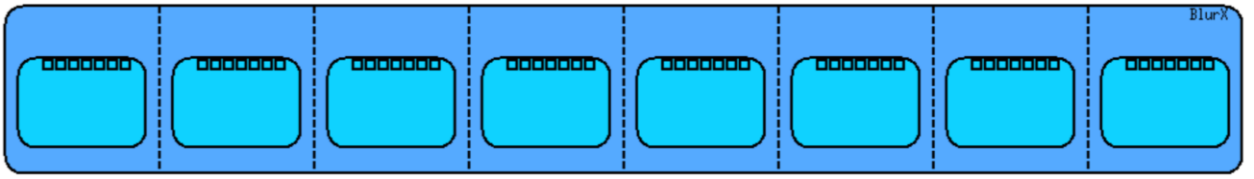

Figure 7: BlurX Unrolling of For Loop

After that, the delays for the BlurX and BlurY modules were updated to 260.5 ms and 273.375 ms respectively. The modifications described above resulted in changes in timings of the instrumented model. The different measurements are shown in table 2. It is important to note the significant decrease in the frame delay and the total simulation time after every optimization.

Table 2: Timing Changes

Model	Frame Delay	Total simulation time
Initial	39,285,000 us	130,950,000 us
After Pipelining	38,821,500 us	98,615,500 us
Separate BlurX & BlurY	24,336,500 us	52,767,500 us
Unloading For Loops	10,983,125 us	28,663,125 us

Performance estimation and throughput optimization

In this final step, the pipelined DUT model was optimized so that the pipeline stages are better balanced and therefore the throughput of the design is improved.

The timing logs produced by the test bench were extended to measure the performance of the model by means of its throughput, i.e. the frames per second (FPS) coming out of the processing pipeline. The frame throughput was observed 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. The FPS results (initial) were noted down to be compared to the improved ones later.

Then the timing delays were re-estimated using more realistic assumptions by assigning suitable processing elements (PEs) to each pipeline stage using the SCE tool. Table 3 shows the mapping between the blocks in the Canny platform on one hand and the allocated PEs on the other.

Table 3: Mapping Canny Blocks to PEs

DataIn	DMA1
DUT	ARM9x10core1
BlurX	ASIC1
BlurY	ASIC2
Magnitude_X_Y	ARM9x10core2
Non_Max_Supp	ARM9x10core3
Apply_Hysteresis	ARM9x10core4
DataOut	DMA2

After performing a more realistic allocation and mapping, the FPS results (realistic assumptions) were noted down to be compared later in table 4 with the results of later optimizations. The SCE profiler was used to estimate and evaluate the performance. Figure 8 shows the estimation results in a bar chart, it is important to note the badly balanced load for the pipeline stages. More specifically, the *non_max_supp (nms)* stage, in bright green, has a very high computation load compared to the other stages (*nms* stage >1.25 s vs other stages < 0.5 s).

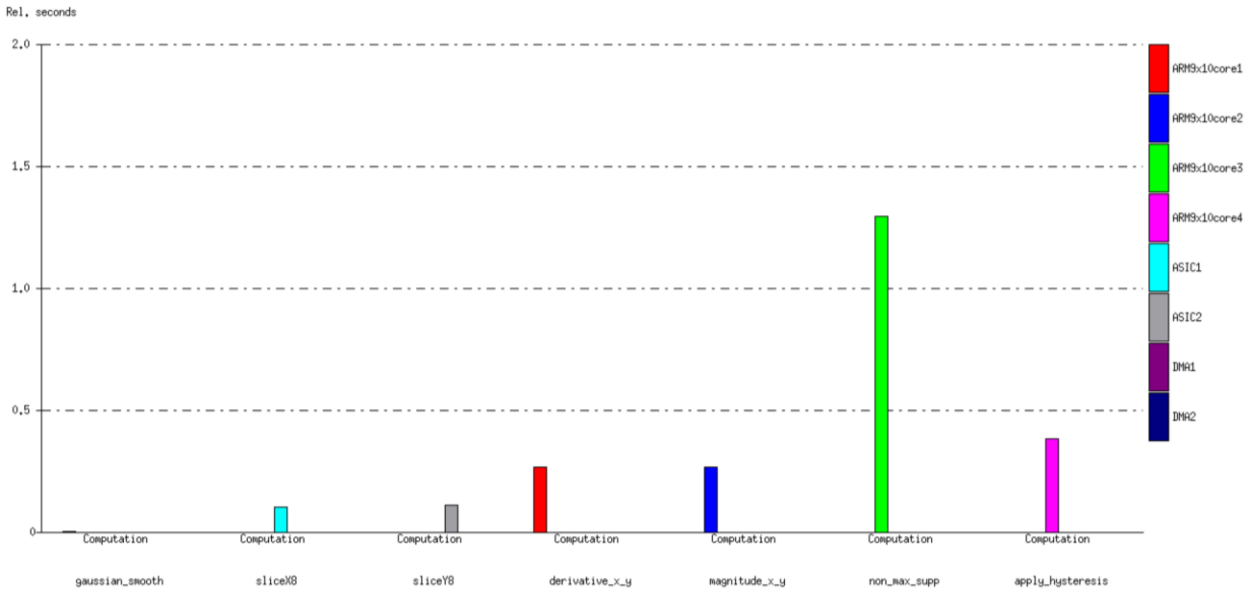

Figure 8: Performance Evaluation of Pipeline Stages

To investigate the reason behind the high load of the nms stage, a pie chart of the ALU operations performed, shown in figure 9, was generated. This was done by double-clicking on the highest (nms) bar, then double-clicking on the ALU operations in the generated intermediate pie chart. Figure 9 shows the immense difference in the ratio of integer (2.9%) to floating-point operations (97.1%). This indicated that the floating-point calculations were one of the significant bottlenecks in the overall performance. This is why these calculations were replaced by faster and cheaper fixed-point arithmetic with acceptable loss in accuracy. In particular, the appropriate shift-operations were added so that integer variables can represent fixed-point values within appropriate ranges.

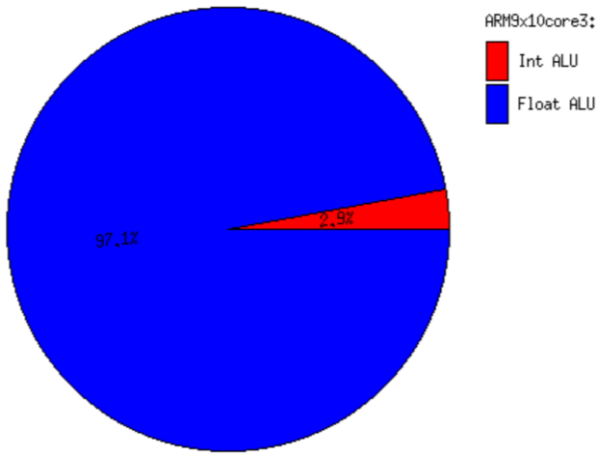

Figure 9: Ratio of ALU Operations in NMS Stage

The ImageDiff tool was then used in order to determine whether or not fixed-point arithmetic was acceptable for the application, the output images quality was compared to the quality of images obtained using floating point operations. The results of the comparisons are shown in figure 10; 65 pixels was the maximum change in the output images, this change is evaluated by the tool as 0.002% and hence was deemed acceptable for the edge detection application.

```
1  mismatching pixels (0.000%) identified in diffEngPlaza001_edges.pgm.
3  mismatching pixels (0.000%) identified in diffEngPlaza002_edges.pgm.
2  mismatching pixels (0.000%) identified in diffEngPlaza003_edges.pgm.
1  mismatching pixels (0.000%) identified in diffEngPlaza004_edges.pgm.
0  mismatching pixels (0.000%) identified in diffEngPlaza005_edges.pgm.
0  mismatching pixels (0.000%) identified in diffEngPlaza006_edges.pgm.
0  mismatching pixels (0.000%) identified in diffEngPlaza007_edges.pgm.
0  mismatching pixels (0.000%) identified in diffEngPlaza008_edges.pgm.
3  mismatching pixels (0.000%) identified in diffEngPlaza009_edges.pgm.
2  mismatching pixels (0.000%) identified in diffEngPlaza010_edges.pgm.
33 mismatching pixels (0.001%) identified in diffEngPlaza011_edges.pgm.
0  mismatching pixels (0.000%) identified in diffEngPlaza012_edges.pgm.
10 mismatching pixels (0.000%) identified in diffEngPlaza013_edges.pgm.
2  mismatching pixels (0.000%) identified in diffEngPlaza014_edges.pgm.
1  mismatching pixels (0.000%) identified in diffEngPlaza015_edges.pgm.
20 mismatching pixels (0.000%) identified in diffEngPlaza016_edges.pgm.
3  mismatching pixels (0.000%) identified in diffEngPlaza017_edges.pgm.
0  mismatching pixels (0.000%) identified in diffEngPlaza018_edges.pgm.
65 mismatching pixels (0.002%) identified in diffEngPlaza019_edges.pgm.
65 mismatching pixels (0.002%) identified in diffEngPlaza020_edges.pgm.
```

Figure 10: Output Images Comparison Results

In order to gauge the timing effects of using fixed-point arithmetic in the non_max_supp calculations, the FPS results (no float operations) were noted down to be compared later in table 4 with the results of later optimizations, and the SCE profiling and evaluation tool was used again with the same allocated PEs and mapping. Figure 11 shows the estimation results of the new computation profile. The workload of the pipeline stages in now better balanced: all the stages now have a computation time less than 550 ms.

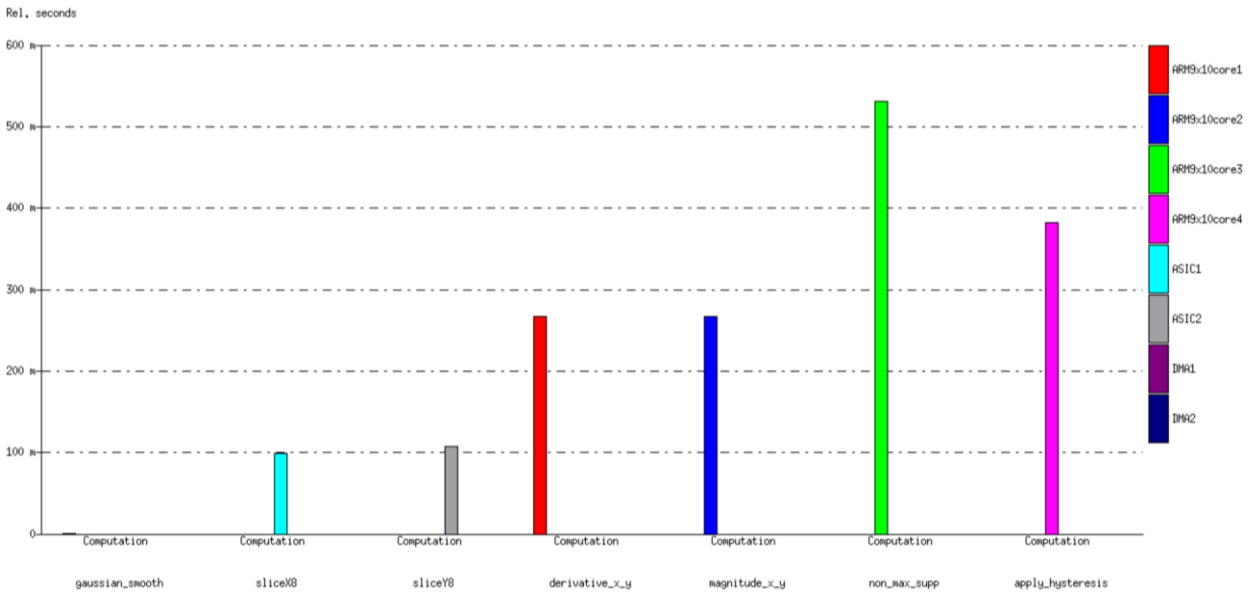

Figure 11: Performance Evaluation of Pipeline Stages after Float Removal

Particularely, the nms stage's load decreased significantly because there are no float ALU operations (0.0%) used anymore as shown in figure 12.

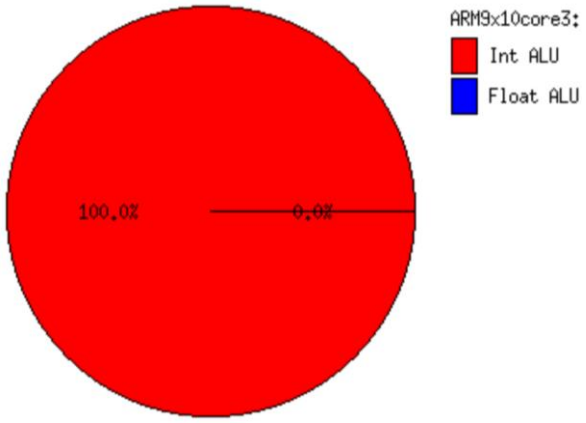

Figure 12: Ration of ALU Operations in NMS Stage after Float Removal

In order to compare the timings shown in the simulation log for each sub-step taken, table 4 presents the different throughputs (FPS) of the pipeline that were noted down previously.

Table 4: FPS Differences and Comparison

Result Name	FPS	Total Simulation Time (us)
Initial	0.735	28663125
Realistic Assumptions	0.769	29890801
No Float Operations	1.117	21800801

The timing log in the "Initial" module differs from that in the "Realistic Assumptions" module. The simulation time increases after performing the PE mapping changes (from 28663125 to 29890801). This is due to the increased initial frame delay, which can be caused by the communication overhead between the ARM and the ASICs.

As expected, the FPS and the total simulation time for the "No Float Operations" step is much better than the previous models (1.117 frames per second compared to 0.735 and 0.769). This improvement is because of the multiple modifications that were described above.

The final result of the application is presented by showing (in figure 13) one of the 20 images that form the video before and after the Canny conversion.

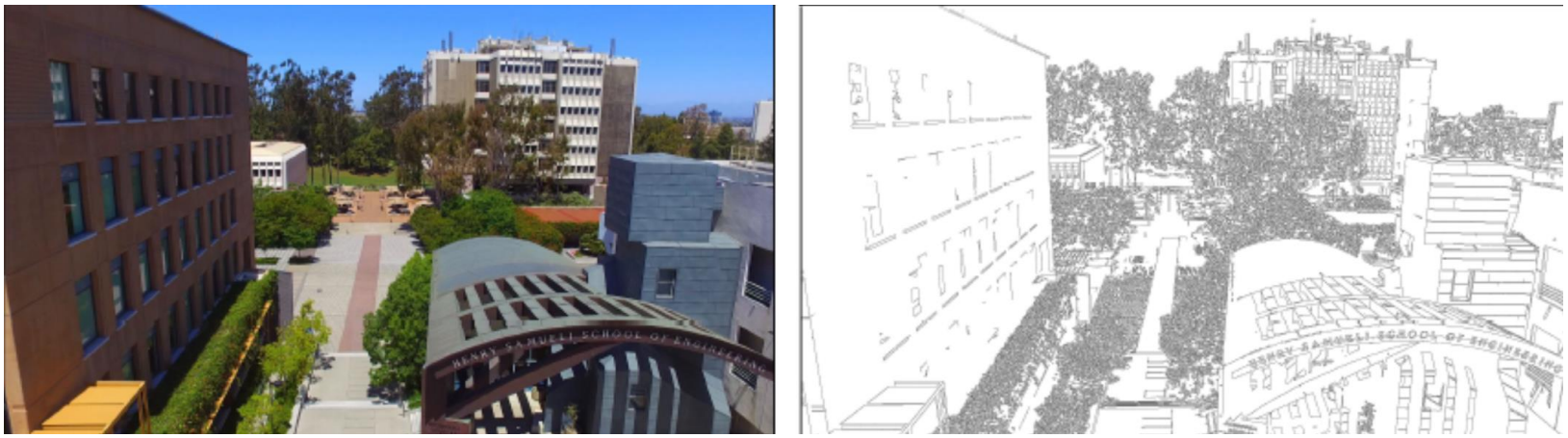

Figure 13: EngPlaza001 Before and After Canny Conversion

The final throughput is still not satisfactory for the real time display of edge images in a video camera since movie cameras use a standard exposure rate of 24 FPS. To further improve the performance, additional modifications such as removing the "float" operations from the rest of the code, using faster or more specialized hardware, parallelization and the use of multicore technology can be implemented. Also, additional changes to the application could be performed and still have the application be acceptable to the end user: proc. For example, the goal can be modified to have the processing and delivery of the edge images take a while before they appear to the user. In other words, more flexibility with the "real-time" constraint can be implemented depending on user requirements. Finally, a lower resolution of pictures can be adapted which would speed up the processing of each image.

III- Summary and Conclusion

In conclusion, all the steps taken in the design of the embedded system model of the Canny Edge Detector were described. All the modeling tasks performed from the starting point of downloading the application source code down to creating the final model obtained were documented. Finally, the background and reasoning for taking the performed steps and design decisions were provided.

Lessons Learned

In this project, I learned that in order to perform optimizations, one should study the application at hand and figure out the bottleneck point then start with the modifications. For example, many optimizations were possible in the different stages of the pipeline. However, since the NMS stage was the bottleneck, the modifications performed on that stage increased the throughput of the pipeline significantly. Moreover, optimizations performed virtually on the application may not translate well and give the wanted performance in the real world. Finally, there are different methods for optimizations when transforming an application from its original C code to a suitable embedded system model. Some of those methods include: increasing the frequency of used processors, implementing a pipeline and make sure the stages are balanced,

parallelizing sections of independent code, and replace complex operations with more simple ones and evaluate the lost precision.

Future Work

Since this work was for a course project, there is a lot of room for growth. More specifically, the modifications and changes discussed in the previous section can be implemented to get an even better frames per second throughput. Ideally, the future work should obtain the wanted application of real-time video handling (processing a sequence of images extracted from a stream of video frames).

IV-References

[1] Wikipedia, "Systems modeling," Wikipedia, the free encyclopedia, 19 Apr 2017. [Online]. Available: https://en.wikipedia.org/wiki/Systems_modeling. [Accessed 2017].

[2] R. Dömer, *EECS 222: Embedded System Modeling | Lecture 2*, Irvine: University of California, Irvine, 2017.

[3] R. Dömer, *EECS222: Embedded System Modeling | Lecture 3,* Irvine: Univeristy of California, Irvine, 2017.

[4] M. Heath and S. Sarkar, *"Canny" edge detector,* Tampa: University of South Florida, 2001.