Modeling of a Canny Edge Detector for Embedded Systems Design

Abstract:

In this course, we design to using the system-level specification, the System-C language, and model to build a project, which can detect the scene by a digital camera on a drone and change the real-time images into the grey-scaled image so that we can get the real-time edge image. To achieve the goal of this project, we first rewrite the dynamic memory allocation in the Canny Edge Algorithm and make them become static. Secondly, we make the seven core stages of Canny Edge Algorithm become System-C module and change its structure into pipelined and paralyzed version. Finally using other methods like adding flags when optimizing or changing the floating-point arithmetic into fixed point etc. to get the best throughput we need.

Introduction

Embedded system modeling and design concepts:

1.Embeded system design advantages and challenges:

Embedded system design has many advantages. For example, we can know the application of the design during the design process.[1] We can also know the application environment and scene of the system (the small size and high reliability of the embedded system can make it apply to a harsh environment). At the same time, the embedded system design can be functionally optimized and run at lower power consumption, the embedded system consumes less power and is more reliable than a general-purpose computer system. In contrast, it also has many disadvantages, such as:

1. Limited system resources
2. Limited processing capacity
3. Higher system reliability requirements
4. More complexity in design

So the embedded system design is also faces many problems and chanllenges

2.Abstract level &Top and down design

An embedded system design faces two problems. The first one is its design complexity. For a lower level of design, it has a large number of components, and we should focus on a mass of designing details. A way to solve this problem is that we can focus on the higher level of abstraction, we can ignore some less important details and work with fewer components to decrease the complexity. However, when the abstraction level is higher, the accuracy is lower. It is the second big problem that we should focus on. A top-down approach, also known as stepwise design, is a good way in embedded system design. It is essentially the splitting of a system to gets the insight part of its sub-system. [2] We begin our design from the top abstraction and move down with increasing of implementation details until our system specification become the basic components. When we put them together, we create the entire system.

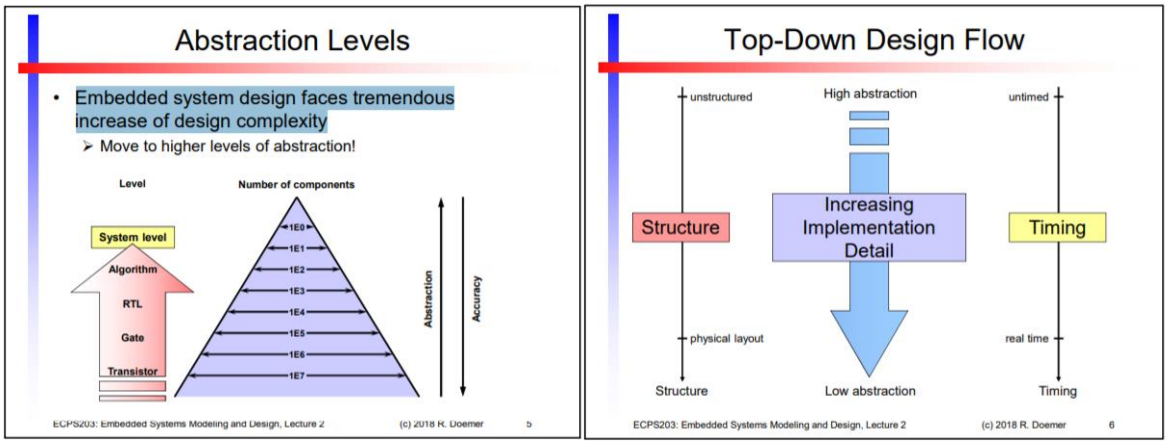

Fig.1 Abstraction Level &Top-Down Design [1]

The IEEE SystemC language:

1. Introduction:

The Wikipedia describe SystemC as: “a set of C++ classes which provide an event-driven simulation. These facilities enable a designer to simulate concurrent processes. SystemC processes can communicate in a simulated real-time environment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as user defined. In certain respects, SystemC deliberately mimics the hardware description languages VHDL and Verilog, but is more aptly described as a system-level modeling language.” [3] SystemC is popular for its plain syntax, its support for exsiting HDL and C modeling and large number of tools.

2. Syntax

The SystemC language is covered by C++, so all of its describes can be made by C++ syntax.

3. Modules and connectivity

When using the SystemC, we should first creat some modules, using the port as interfaces to communicate with other channels or ports. The port is an element of connectivity, which connect the data from inside modules to outside. The channel is a fifo or other types bus to make the port connect and transmit values.

4. Process and events

SystemC languege alows us to sychronize process and event, which should be defined during initialization. [3]

5. Reason to use

In this project, we will use the pipeline and parallel to refine our test bench, so we choose to use the SystemC language whose feature can of great use.

Case study of a Canny Edge Detector for Real-time Video

Structure of the Canny edge detection algorithm:

Before we start our own process, we initially read and analysis the canny source code in order to understand it well. So the first step we did is write a function call tree and find the structure of the Canny edge detection algorithm. We broke it into 5 levels as the picture shows.

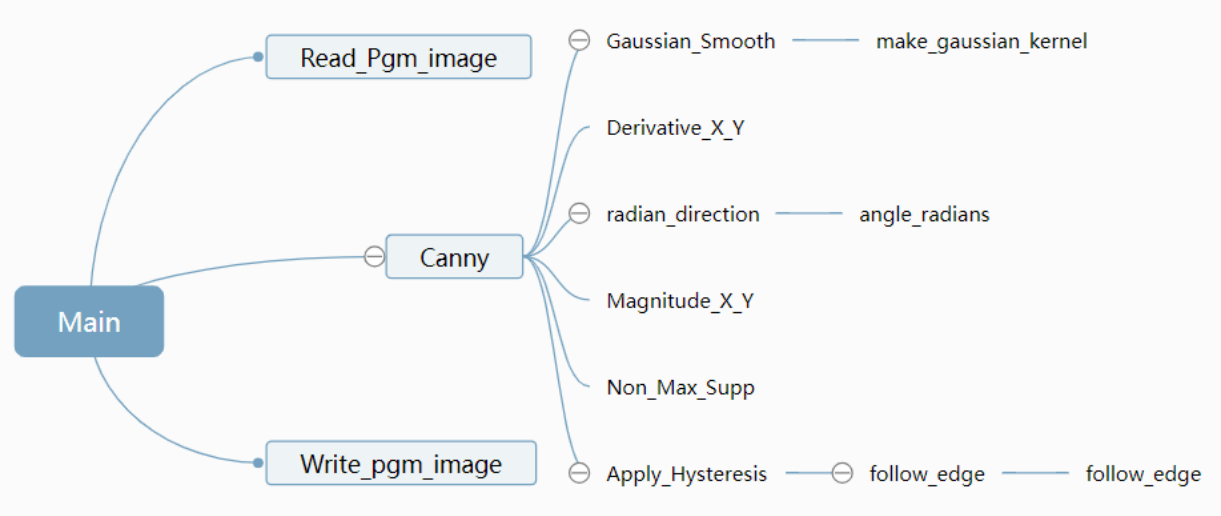

Fig.2 Function call tree

The Read_Pgm_image is used to read PGM format image from a file or standard input, and the Canny is the Canny Edge Algorithm code which is used to process image. Then the Write_Pgm_image function makes outputs in PGM format. In the Canny module, there are 6 function modules: The Gaussian_Smooth() performing smoothing process on the image to reduce the noise. The Derivative_X_Y() compute the first derivative in the x and y direction. And because not all the derrivatives are compute in the same way. I need a radian_direction function to compute the direction of a gradient image from dx and dy image. The next function Magnitude_X_Y() is used to magnitude the gradient images. Non_Max_Supp() is used to make non-maximal suppression to the magnituded image. Finally, the Apply_Hysteresis() is used to find edges.[4] By writing this function call tree, I can find the relationship between these functions, and gain a deeper understanding of this algorithm. It helps me decide how to optimize the code and make it suitable for hardware.

Modeling and simulation in IEEE SystemC:

1. Clean C++ model with static memory allocation:

Because we will finally use the SystemC language to simulater my project, we used C++ as modeling language and clean the C++ model so that there is no warnings and no errors during compilation. we used the g++ compiler with -Wall -pedantic -O2 options to find more irregular code and fix them which may pay off in later process. Moreover, due to the hardware environment, we cannot set any flexible software configuration parameters but changed them into hard-code constant so that it can run correctly and fast on chips.[5] And then we removed the dynamic memory allocations such as malloc() and calloc() functions, and wrote static structure instead. As we all know, a hardware cannot instantiate a new memory during the run time, so we should fix the memory size during the compile time.

2. From single image to video stream

To finally make the real-time Canny Edge detector, we should process a video stream. That means we should process a stream of images using Canny Edge Algorithm in real time. In order to achieve this goal, we extracted 30 video frames from the drone movie and process these sequence of frames as a pioneer. In this step, we met a problem in the Linux environment—— segmentation fault. Because we were using the image from drone camera, the image pixel is changed to a higher resolution **2704*1520**, which is different from the initial **320*320**. I should use the ***ulimit -s 12800*** or *limit* ***stacksize 128 megabytes*** command to avoid the stack crash. This is because larger image leads to higher memory usage and we were using the array to save the image data and all of these array variables are local variable getting allocated on stack. There is no doubt that changing image size will cause the stack limitation. [6]

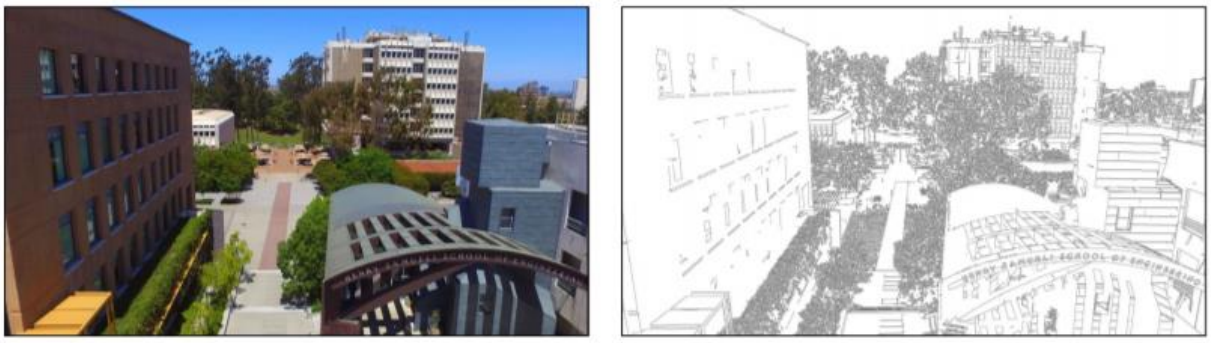

Fig.3 Video stream [7]

3. Create a Test bench model

In this part, we created a test bench and platform structure in System-C model. As the Fig.4 shows. This structure includes a top-level module Top and inside it there are three modules: Monitor, Platform and Stimulus module. The Stimulus module runs 30 times to output 30 images to the Platform, and in Platform, the DUT runs image process Canny. Finally the Monitor module reveives the processed image and make it output file. The most confusing thing in this step is the DataIn and DataOut module, which seem to be useless. However, it is

actually of great use when using detailed bus protocol [8] to communicate the DUT module, because it allows our test bench stay unmodified.

```
Top top
|------ Monitor monitor
|------ Platform platform
|       |------ DUT canny
|       |------ DataIn din
|       |------ DataOut dout
|       |------ sc_fifo<IMAGE> q1
|       \------ sc_fifo<IMAGE> q2
|------ Stimulus stimulus
|------ sc_fifo<IMAGE> q1
\------ sc_fifo<IMAGE> q2
```

Fig.4 Test bench structure [8]

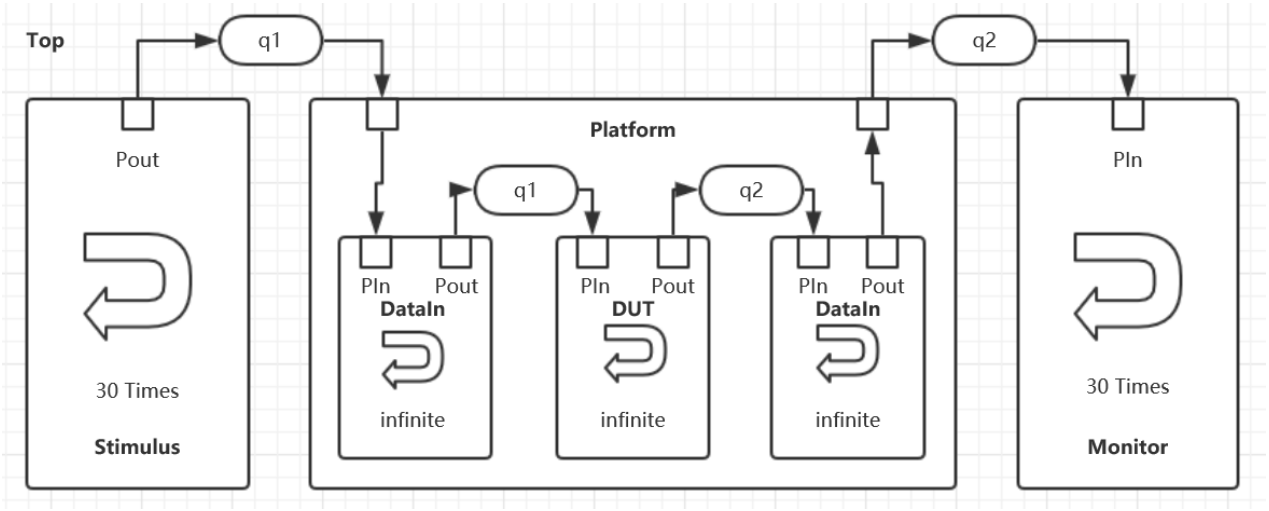

Fig.5 Test bench model

4. Structure the DUT and Canny profiling

We then tried to structure the DUT and the Canny profiling. As the Fig.6 shows, the function in Canny should be refine as systemC modules also. This is because all these fuctions should be well revised as sc_module in suitable structural hierarchy for our later use. We tried to test our model and to get useful informations about the time usage of each module so that we can continue refining our test bench.

```
Platform platform
|------ DataIn din
|------ DUT canny
|       |------ Gaussian_Smooth gaussian_smooth
|       |------ Derivative_X_Y derivative_x_y
|       |------ Magnitude_X_Y magnitude_x_y
|       |------ Non_Max_Supp non_max_supp
|       \------ Apply_Hysteresis apply_hysteresis
\------ DataOut dout
```

Fig.6 The structure of DUT [8]

Refine test bench by pipelining and parallelization:

1. Structure the DUT and Canny profiling

The next step is testing the first refined test bench. we used the gprof tools, ran gprof Canny and got a Flat profile with other logs we didn't use on the screen. The Flat profile showed the function modules' time usage. we recorded these time datas and calculated them into percentage with scale of 100%. As the Fig.7 shows, the percentage of the time use per sencond of these 5 function-modules and their sub-modules.

```
Gaussian_Smooth 42.9%
|------ Gaussian_Kernel 0%
|------ BlurX 19.46%
\------ BlurY 23.49%
Derivative_X_Y 5.4%
Magnitude_X_Y 13.9%
Non_Max_Supp 23.3%
Apply_Hysteresis 14.5%
                 100%
```

Fig.7 My testing result of function time usage

2. Record the real-time running time in Pi

From this figure we can easily find the most time consuming module, which seems like to be the first bottleneck of our test bench, the Gaussian_Smooth module. So, the next point is how shall we deal with the BlurX and BlurY process inside Gaussian_smooth module. Meanwhile, we should not only see the test simulation on our linux sever, but also focus on the real-time performance of our model on hardware because the real-time performance may totally different from it on simulator, due to the differences in the CPU, the threads, the clock frequency and so on. To make this, we choose to run our code on raspberry Pi and measure these datas on real board. We used the assignment4's source code, and run it on the Respberry Pi. I made each

function-call between clock_t startTime and clock_t endTime, and than we can calculate the calling time of each function. We set a total number variable to store the datas of 30 times, and than make it over 30 to get the average run time result. The result of this realtime shows on the Fig.8.

```
derrivative_x_y run time = 0.404407 secs
magnitude_x_y run time = 0.875691 secs
non_max_supp run time = 0.669334 secs
apply_hysteresis run time = 0.554707 secs
kernel run time = 0.000013 secs
blurx run time = 1.444174 secs
blury run time = 1.690140 secs
```

Fig.8 The run time of functions on Raspberry Pi

3. Five steps to improve the running performance

We can see the blurX and blurY is actually the bottleneck of running speed. At Assignment8, we created 5 steps to improve our running performance.

Step1&2 Add the frame delay to the simulation code. When we choose to instrument the model with logging of simulation time and frame delay, we can see a fun thing that on the server's print screen, all process have 0 time delay as the Fig.9 shows. This is because on our linux sever simulator, the simulation time is always 0 and there is no delay between any frames. So we should add the real-time delay manully.

```
    0 s: Stimulus sent frame  1.
    0 s: Stimulus sent frame  2.
    0 s: Stimulus sent frame  3.
    0 s: Stimulus sent frame  4.
    0 s: Stimulus sent frame  5.
    0 s: Stimulus sent frame  6.
    0 s: Monitor received frame  1 with      0 s delay.
    0 s: Stimulus sent frame  7.
    0 s: Monitor received frame  2 with      0 s delay.
[...]
    0 s: Stimulus sent frame 30.
    0 s: Monitor received frame 23 with      0 s delay.
    0 s: Monitor received frame 24 with      0 s delay.
    0 s: Monitor received frame 25 with      0 s delay.
    0 s: Monitor received frame 26 with      0 s delay.
    0 s: Monitor received frame 27 with      0 s delay.
    0 s: Monitor received frame 28 with      0 s delay.
    0 s: Monitor received frame 29 with      0 s delay.
    0 s: Monitor received frame 30 with      0 s delay.
    0 s: Monitor exits simulation.
```

Fig.9 The log of frame delay [9]

For the convenience, we made the rule that everybody use the same delay time as the Fig.10 shows. We wrote these delay time into each module, as wait(delaytime) and then simulate.

```
Receive_Image              0 ms
Make_Kernel                0 ms
BlurX                   1710 ms
BlurY                   1820 ms
Derivative_X_Y           480 ms
Magnitude_X_Y           1030 ms
Non_Max_Supp             830 ms
Apply_Hysteresis         670 ms
```

Fig.10 Frame delay time[9]

Step3 Refine the log statement. We finally measured the total run time, the delay between stimulus and monitors and the FPS result we need. By refining our log statement, we print them on the screen to see how this application performs at this situation.

```
[...]
55680 ms: Monitor received frame 28 with 15860 ms delay.
55680 ms: 1.820 seconds after previous frame, 0.549 FPS.
57500 ms: Monitor received frame 29 with 15860 ms delay.
57500 ms: 1.820 seconds after previous frame, 0.549 FPS.
59320 ms: Monitor received frame 30 with 15860 ms delay.
```

Fig.11 Extra log refinement result[9]

Step4 Pipeline the DUT into 7 stages

Because our work was already in pipeline, we just need to make sure that there is no cross-stage channel and make all the buffer size stay as 1. As a result, the Fig.12 is the final version of the test bench, the receiver and kernel are the first stage, the blurX is the second one, blurY is the third one, from Derrivative_x_y to Apply_hysterisis are the lasts.

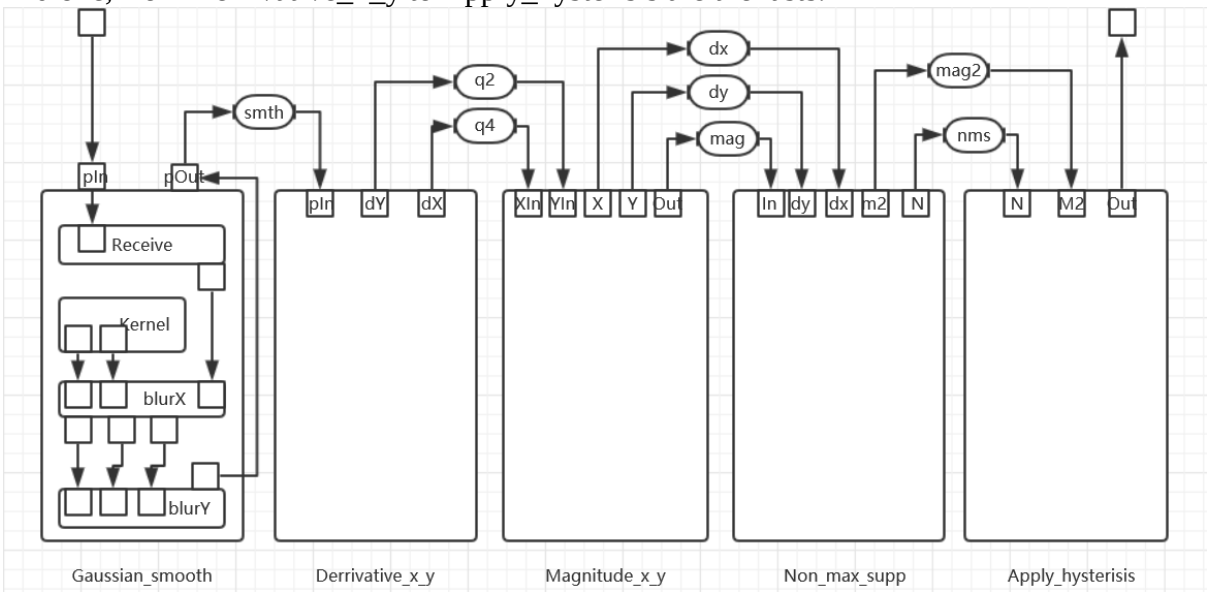

Fig.12 The pipelined test bench model(Omitted showing the Channel in Gaussian_smooth)

Step5 Solve the bottleneck problem

As I said before, the blurX and blurY in Gaussian_smooth is a bottleneck problem, which means the key point to increase the speed of running is to overcome it. So that is why we tried to find the way to deal with this problem. The best way to do so is parallelizing the blur module. Theoretically, we can parallel every cols and rows of a image by using GPU, however at this project, we may use the Raspberry Pi as the processor, so we sliced the blurX and blurY into 4 parallel parts by using 4 different threads, which depends on the featrues of the Pi. The speed was expected to increase about 4 times.

Model Frame	Delay	Throughput	Total simulated time
CannyA8_step1	0 ms	/ FPS	0 ms
CannyA8_step2	6540 ms	/ FPS	59320 ms
CannyA8_step3	6540 ms	0.549451 FPS	59320 ms
CannyA8_step4	6540 ms	0.549451 FPS	59320 ms
CannyA8_step5	3892 ms	0.970874 FPS	33762 ms

Fig.13 Result of each step

As far as this result is concerned, step5 is not 4 times quicker than step4. It throwed us a doubt why this parallelizing didn't work? The answer is that the simulator just has a single thread, and honestly it cannot simulate in parallel. If we run our code on a ASIC or GPU which has support of multi-threads, we will get the right result.

Performance estimation and throughput optimization:

This two picture shows the time result before and after parallelize the blur.

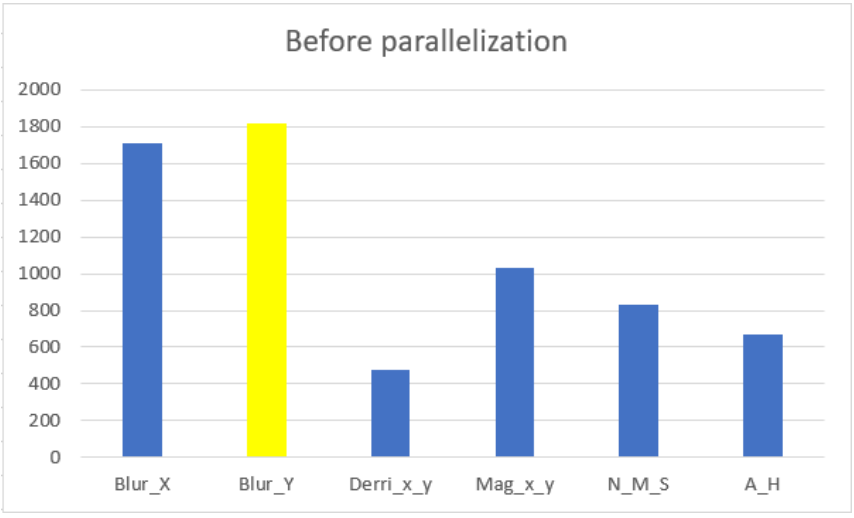

Fig.14 Bottleneck before parallelization

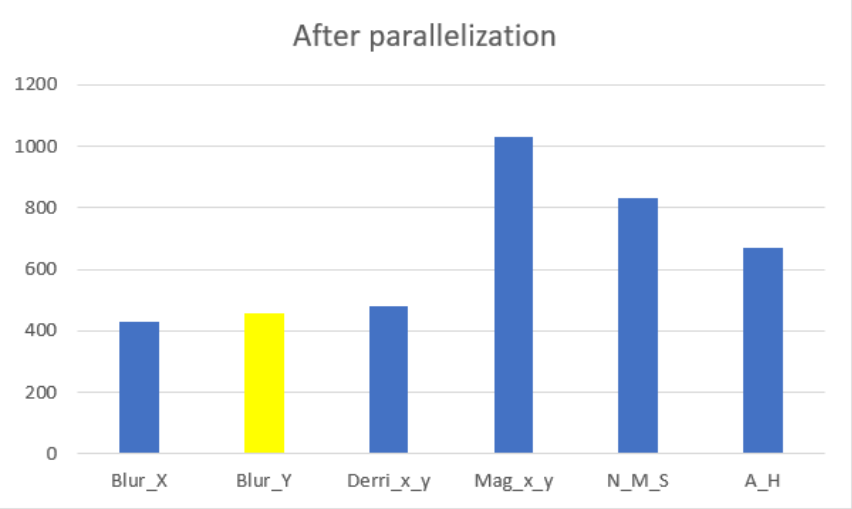

Fig.15 Bottleneck after parallelization

Comparing this two picture, we can see after we finishing parallelization, the bottleneck was changed from blur to the magnitude_x_y and we need to make our model further optimized. From this way, we had two steps:

Step1 Turn on compiler optimizations for maximum execution speed

The GNU compiler gives us many different compile options such as -O(2-12), -mfloat-abi=hard, -mfpu=neon-fp-armv8 that we can choose them to maximum our execution speed. [10] To me, I just use the -O3 flag when I was compiling my model on Pi. I got optimized data as follows:

```
step1:
if not change the floating point to fixed point, just using -O3 flag
derrivative_x_y run time = 0.131436 secs
magnitude_x_y run time = 0.129954 secs
non_max_supp run time = 0.230948 secs
apply_hysteresis run time = 0.176484 secs
kernel run time = 0.000014 secs
blurx run time = 0.201795 secs
blury run time = 0.391997 secs
```

Fig.16 The optimized results ran on Pi

We back-annotated this result into our model, and then we got an optimized result where the FPS had been improved up to 4.329:

Model Frame	Delay	Throughput	Total simulated time
CannyA9_step1	815 ms	4.329 FPS	7514 ms

Fig.17 Back-annotate result

Step2 Consider fixed-point calculations instead of floating-point arithmetic

According to Fig.16, the non_max_supp is what we should focus on next except the blurY. In Non_max_supp function, there exists 2 floating-point calculations. As is know to us all, the fixed-point calculation is much faster and cost less than floating-point calculation, so we tried to change these 2 floating-point calculations into fixed-point even if this action may lower the quality of image. The Fig.18 is my result by changing the float-point into fixed-point.

```
step2:
if change the floating point to fixed point
derrivative_x_y run time = 0.133413 secs
magnitude_x_y run time = 0.132099 secs
non_max_supp run time = 0.559704 secs
apply_hysteresis run time = 0.178834 secs
kernel run time = 0.000017 secs
blurx run time = 0.209051 secs
blury run time = 0.416964 secs
```

Fig.18 Fixed point result

In contrast to our expectation, we can see non_max_supp run time increased to 560 ms, and the FPS decreased to 1.786. The comparation is as follows:

Model Frame	Delay	Throughput	Total simulated time
CannyA9_step1	815 ms	4.329 FPS	7514 ms
CannyA9_step2	1160 ms	1.786 FPS	17400 ms

Fig.19 Two steps'comparation

According to this result, the using of fixed-point arithmetic may decrease the speed in Pi, so I choose to not use the fixed-point arithmetic in my project, and the 4.329 FPS is the final FPS throughput of my work.

We should also note that the relative time differences between A6 and A7 is caused by the differences in the CPU, the thread number, the core number, the clock frequency and many other factors between Pi and simulator. And also, A9 was added to some real-time delay, and using the optimization compiler, so the A9's result is totally different from A6's result.

Real-time video performance:

As far as the final results of this work is concerned, we didn't achieve the 30 FPS but just got a 4.39 FPS. This can not achieve real-time video to user. However, for the real-time aspect, we may not actually need the 30 FPS, I think 15 FPS is also to show the video stream of canny edge images. On the other hand, there is no need to show such big size images to user, we can make them smaller and then get the higher FPS.

Summary and Conclusion

Lessons Learned:

The biggest gain to me is that I have learned a lot of things about embedded system design, IEEE SystemC language and other theories related to them that I never learned before which can be appllied to my original major area—Structure Engineering. I learned that the dynamic memory allocation is not suitbale for hardware, I learned how the pipeline and parallel can contribute to higher speed and I also learned the theory of Top-down designing. From details to abstracts, I experienceed a lot of embedded system design. I think all of these things will support me in the future work to build some area-related systems and make a difference.

From this course, I first learned the concepts of embedded system modeling. I got a total view of how to design an embedded system including abstract theory, hardware features, the model structure, the communication, the interfaces and sychronization. And then we have a deeper understanding of IEEE SystemC language through the weeks, which is very important in modeling our system in achieving these concepts above. By doing assignment of Canny Edge, I have the practice of pipelining and paralization and practice how to estimate an embedded system from different aspect. That is very useful for our future designing.

In details, for example, due to the single thread of the simulator, I found the paralization doesn't work in assignment 8. This result helps me learn that optimization actions may really constrained by the hardware devices, and if I choose to use the advanced devices like GPU or another, I may cost a lot. So as a embedded system designer, how to balance the function and cost is also an important thing.

Finally, as a system designer, I learned that simulation is an essential part during the design. I could not try our untested model on chip directly, which may lower the efficiency or even destroy the devices to make unnecessary cost.

Future work:

Initially, for this project, I finally got a result of 4.329 FPS, it is far less than getting a real-time video. So, in the future, I could continue to improve our works as follows:

1. Use the GPU to speed up the blur modules.
2. Make the image size smaller, such as shorten ½ of rows and cols, the speed will improved by 4 times, or even shorten ¾ of rows and cols so that I can get about 16 times speed up.

Moreover, I used some tools to change png image into pgm image, and then make them grey-scaled manully. When I create the drone detector or other application, I should add some codes or algorithms in order to directly shifting the image type from png to grey-scaled pgm on hardware.

References:

[1]D. Rainer, "Lecture1", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].

[2]"Problem Solving: Top-down design and Step-wise refinement - Wikibooks, open books for an open world", En.wikibooks.org, 2018. [Online]. Available: https://en.wikibooks.org/wiki/A-level_Computing/AQA/Problem_Solving,_Programming,_Data_Representation_and_Practical_Exercise/Problem_Solving/Top-down_design_and_Step-wise_refinement. [Accessed: 09- Dec- 2018].

[3]"SystemC", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/SystemC#cite_note-2. [Accessed: 09- Dec- 2018].

[4]" 'Canny' edge detector code", Mike Heath, 1996. [Online]. Available: ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src. [Accessed: 09- Dec- 2018].

[5]D. Rainer, "Assignment 2", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].

[6]D. Rainer, "Assignment 4", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].

[7]D. Rainer, "Lecture 19", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].

[8]D. Rainer, "Assignment 5", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].

[9]D. Rainer, "Assignment 8", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].

[10]D. Rainer, "Assignment 9", DHB 1200 UCI, 2018. [Accessed: 09- Dec- 2018].