UCI University of California, Irvine

Modeling of a Canny Edge Detector for Embedded Systems Design

Course title:

Embedded System Modeling & Design

Course Code: 16905

Haoyu Zhou

Dec 09 2018

[Abstract] In this project, we will create an embedded system model about image processing. Using the IEEE SystemC, we will realize Canny, an Edge Detection algorithm. To optimize program, we use static memory allocation to adapt to embedded devices, separate the program to hierarchy modules to follow a top-down design concept, realize paralleling and pipeline to improve performance. I am trying to use FPGA technology to accelerate throughput of image processing in follow-up experiment in Pynq platform. Next, I will introduce the canny algorithm in detail, the steps I realize the program and compare performance promotion between every step.

[Key words] Image processing; Edge detection; SystemC; FPGA; Pynq

Content

I. Introduction

i. Embedded System Modeling and Design Concepts

1.The Structure of Cyber-Physical System

Cyber-physical systems (CPS) are a broad range of complex, multidisciplinary, next-generation engineered systems that integrate embedded computing technologies (sensors and actuators) and computational algorithms into the physical world[1], like Figure 1.

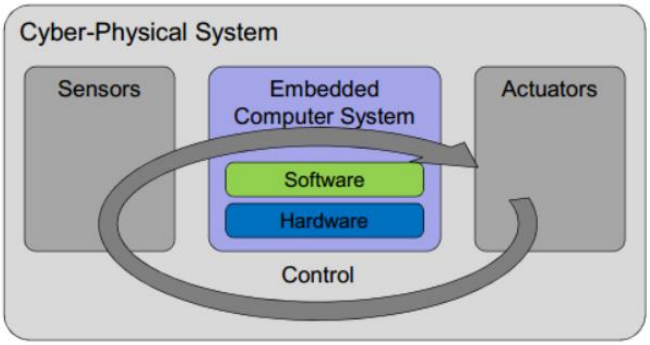

Figure 1

2.Top-down Design Flow

When we want to design our own cyber-physical system, we will face a choice between top-down design flow and bottom-up design flow. Considering the complexity of bottom level, we want to design a general model ignoring some detail at first, and we also want to verify our algorithm in abstract level, therefore top-down design flow is better than bottom-up design flow in that case, like Figure 2.

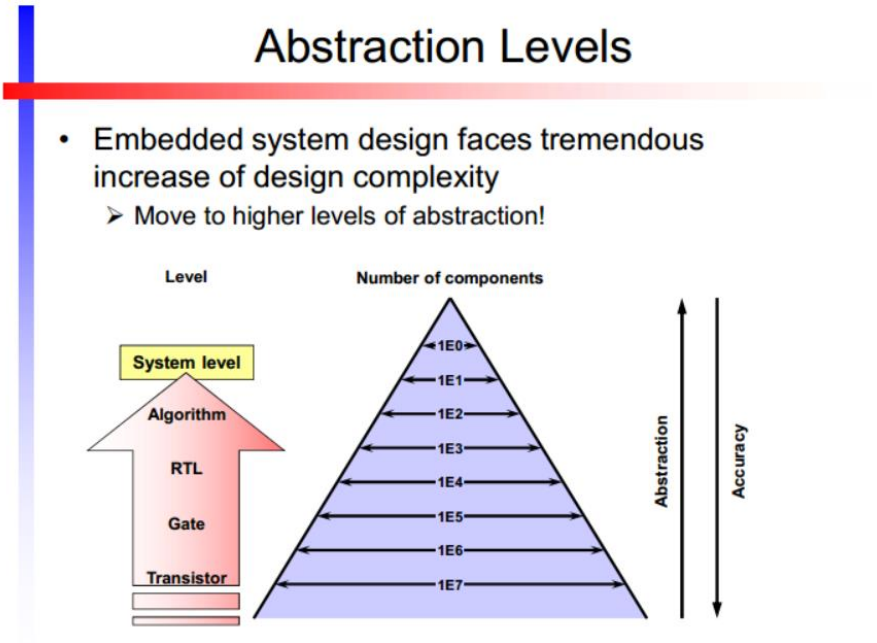

Figure 2 (c) 2018 R.Doemer

[1] Cyber physical systems: https://en.wikipedia.org/wiki/Cyber-physical_system

ii. The IEEE SystemC Language

1.Comparation between several language

To a system-level description language, we hope it could have some specific features aimed at our requirement, such as formality, executability, modularity, simplicity and so on.

Making a comparation between them, shown in Figure 3, SystemC is a popular system-level description language nearly satisfying our whole requirements.

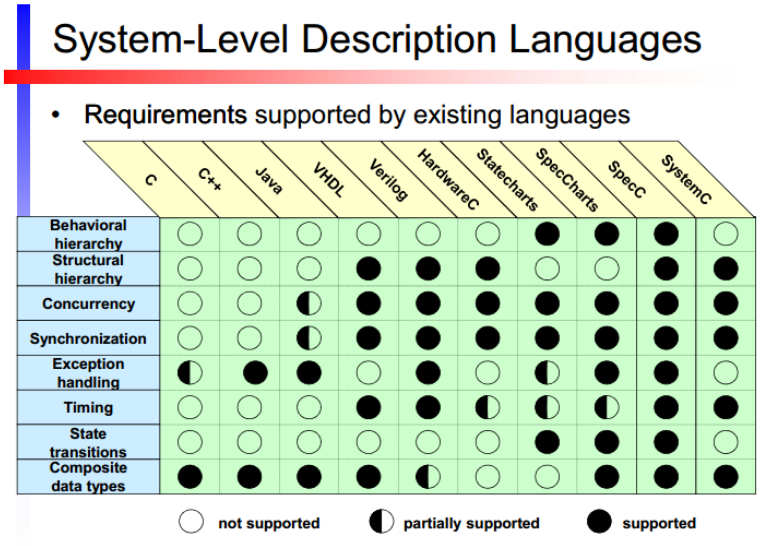

Figure 3 (c) 2018 R.Doemer

2.SystemC Separation of Concerns

Unlike other programming language, SystemC separates computation and communication from system model to implementation model. Shown in Figure 4, yellow one is channel contain green ones, signal, then modules have ports connected to channel. After synthesis, signal will be explored outside.

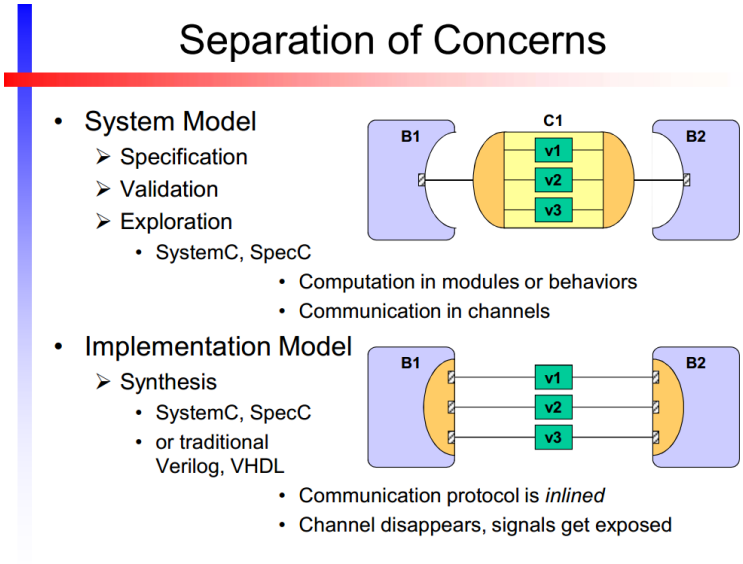

Figure 4 (c) 2018 R.Doemer

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

i. Structure of the Canny edge detection algorithm

1. Background of Canny Algorithm

Canny is a very popular edge detection algorithm developed by John F.Canny in 1986[2]. This algorithm mainly consists of five parts, gaussian smooth, calculating gradients of the image, non-maximum suppression, double threshold to determine potential edge and hysteresis. Figure 5 shows the call tree of Canny.

```
main()
        |- read_pgm_imgae()
        |- canny()
                |- gaussian_smooth()
                        |- make_gaussian_kernel()
                |- derrivative_x_y()
                |- magnitude_x_y()
                |- non_max_supp
                |- apply_hysteresis()
                        |-  follow_edges()
        |-  write_pgm_image()
```

Figure 5

After applying canny to a grey-scale picture, we can transfer Figure 6 shot by the drone to Figure 7.

Figure 6

[2] Canny C++ source code: http://marathon.csee.usf.edu/edge/edge_detection.html

Figure 7

Canny is one of the best edge detection algorithm, the founder of Canny also proposed three classic rules[3] for edge detection.

For such an efficient algorithm, we surely want to use it in embedded system. Although Canny is not fairly complicated, however, for a quite complex and huge program, it's less possible to make balance between effect and efficiency of development and validity when putting it on SoC by tradition way. I think Canny is a good example to learn how to solve it.

2. Detail of Canny algorithm Realization

The original Canny algorithm coded by C++ we can get from this website.

Firstly, we need to fix the user-adjustable configuration parameters for embedded system design. In order to implement our application model later into an actual embedded system, we need to decide on the configuration parameters which were kept flexible in the initial software code. The main parameters we want to fix are the size of the image, sigma deciding the size of the gaussian kernel window, thigh & tlow which are double threshold.

Second, we need to remove all the dynamic memory allocation (i.e. the use of functions malloc(), calloc(), and free()). Due to limitation of memory size and CPU performance, some the desired SoCs cannot instantiate a new memory chip at runtime without operation system supporting, that means the operation system may not have dynamic memory allocation algorithm.

①Gaussian_smooth

In gaussian smooth stage, we need to create a kernel first. `make_gaussian_kernel` function has parameters of sigma, kernel and windowsize. It actually creates a window, and the values in the window are got from gaussian distribution.

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right)$$

[3] Canny edge detector: https://en.wikipedia.org/wiki/Canny_edge_detector

Like Figure 10, it is sigma = 1.4, window size is 5*5.

$$\frac{1}{115}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

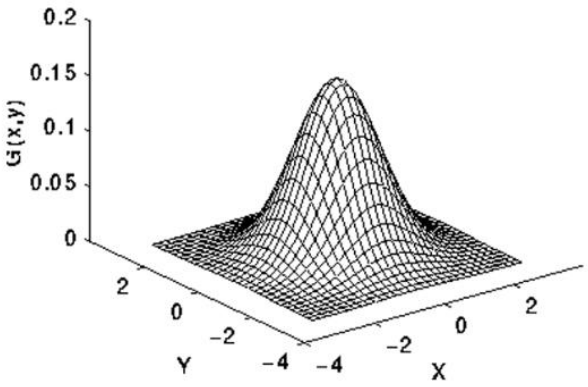

Figure 8

Gaussian smooth removes the noise pixels in the picture, because the noise pixels are totally different from their neighbors, they are isolated. We need a window to calculate every pixel value by its neighbors. For the pixel P, Sigma value decides the neighbors which are far from P have how big influence on P.

```
*windowsize = 1 + 2 * ceil(2.5 * sigma);
```

Now for the code, 1 makes sure the result is odd, 2* means we have left and right, or up and down, neighbors of P, 2.5 is a good choice, because the 95% value in 2*sigma, 99.7% value in 3*sigma range according to gaussian distribution. When sigma is equal to 4 or more, that means the far neighbors of P have big weight, P value = ∑(weight*neighbors value). Therefore, to make sure we do not lose the important neighbors value when sigma is big, we choose windowsize = 21. If sigma is small, it also makes sense, because the far neighbors' weight is small with very small influence near to 0, like Figure 9.

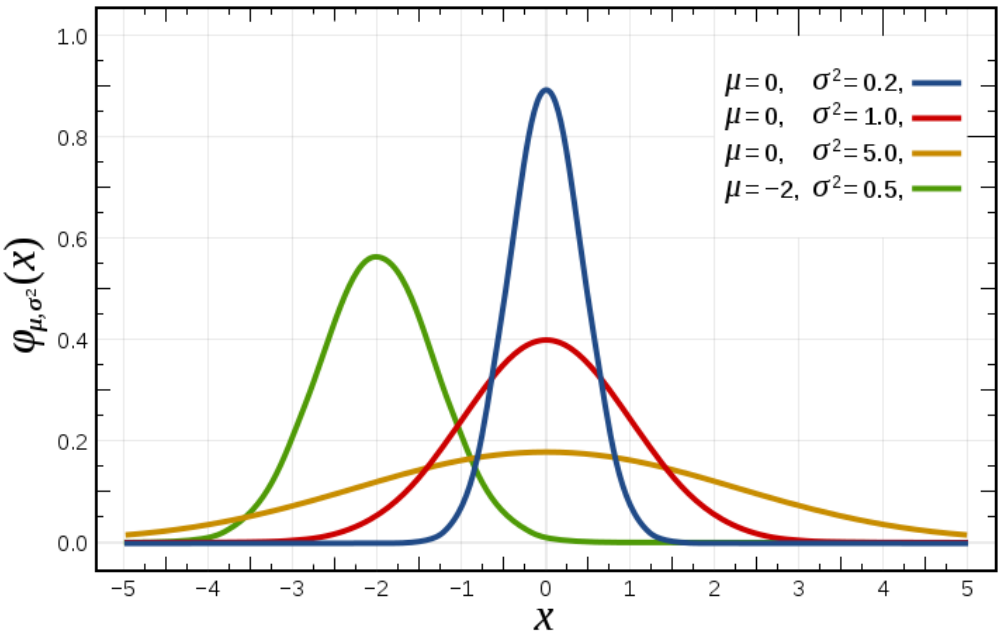

Figure 9

②Derrivate_x_y & Magnitude_x_y

This stage we will find the intensity gradient of the image. In `derrivate_x_y`, we compute the first derivative of the image in both the x any y directions, get

$$\Delta x = x_{i+1} - x_i \text{ and } \Delta y = y_{i+1} - y_i.$$

Then get the magnitude of the gradient

$$G = \sqrt{G_x^2 + G_y^2}.$$

Here, we can set BOOSTBLURFACTOR value in order to make bigger difference.

③Non_max_supp

In this stage, our goal is to find the largest edge to "thin" the edge. At the beginning, we treat the edges of the image as not including pixels we need and we give all of these pixels label of NOEDGE. The direction of one pixel can be got by

$$\theta = \arctan\frac{G_y}{G_x} .$$

Then we let all the pixels which is not the local maximal become NOEDGE. The algorithm for each pixel in the gradient image is:

1. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
2. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y-direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

But in the normal situation, we do not have continuous gradient directions, here, we use a window to make it come true. That means we have eight directions for one pixel and compare every pixel to its contiguous neighbor gradients, like Figure 10. Remember the gradient is vertical to the direction of edge.

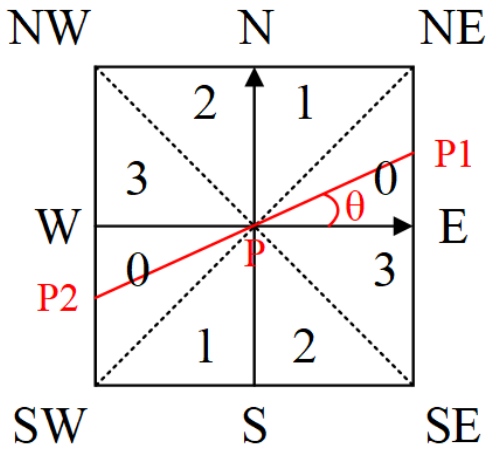

Figure 10

Shown in the Figure 11,

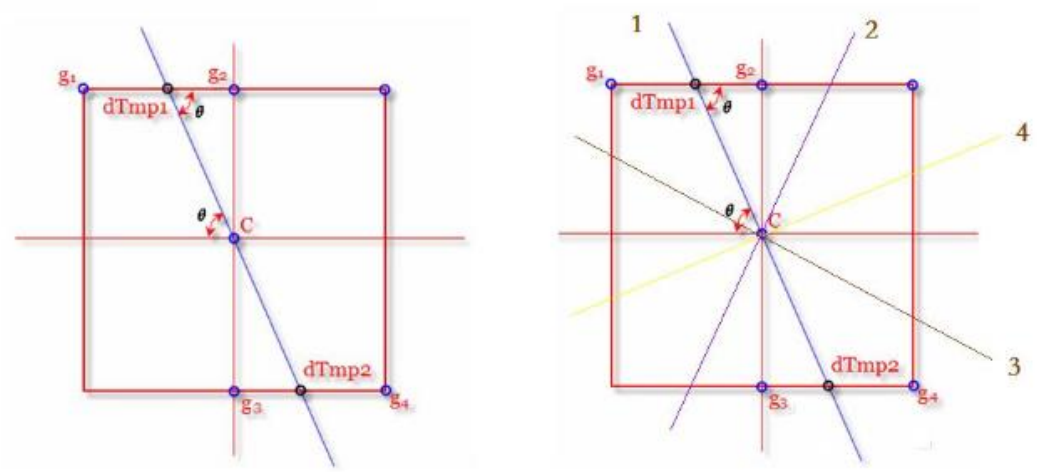

Figure 11

g1, g2, g3, g4 are real pixels in the picture, the blue line is gradient of the pixel we calculate. If the pixel is the local maximal, its gradient is supposed to be large than dTmp1 and dTmp2. However, dTmp1 and dTmp2 are not real pixels in the picture, the way we get the gradient of dTmp1 is calculating the radio of dTmp1, g1 and g2. Let M1 be g1's magnitude, M2 be g2's magnitude and Md1 be dTmp1's magnitude, then we get

$$M_{d_1} = w \cdot Mg_2 + (1 - w) \cdot Mg_1,$$

$$w = \frac{|d_1 - g_2|}{|g_1 - g_2|}$$

If dg is not large than dTmp1 either dTmp2, it is not the local maximal, then it should be label by NOEDGE. If not, POSSIBLE_EDGE.

④Apply_hysteresis

This is a quite interesting stage, we use double thresholds to remove the influence of "weak edges", which are caused by noise and color variation. How to tell the strong edges and weak edges? If the edge pixel's value is higher than high threshold value, it should be label EDGE, if not, POSSIBLE_EDGE.

We use THIGH to determine high threshold value,

```
for(r=0;r<32768;r++) hist[r] = 0;
    for(r=0,pos=0;r<rows;r++){
        for(c=0;c<cols;c++,pos++){
            if(edge[pos] == POSSIBLE_EDGE) hist[mag[pos]]++;
        }
    }
for(r=1,numedges=0;r<32768;r++){
    if(hist[r] != 0) maximum_mag = r;
    numedges += hist[r];
}
highcount = (int)(numedges * thigh + 0.5);
r = 1;
```

```
numedges = hist[1];
while((r<(maximum_mag-1)) && (numedges < highcount)){
    r++;
    numedges += hist[r];
}
highthreshold = r;
lowthreshold = (int)(highthreshold * tlow + 0.5);
```

32768 is 256*128, for the magnitude, we get this from a step of square root calculating. 256 is the 8th power of 2, because the grep-scale pixel is 8 bits, so the size of array named hist is 32768 to collect all probable the value of the magnitude. Finally, we compute the high threshold value as the (100 * thigh) percentage point in the magnitude of the gradient histogram of all the pixels that passes non-maximal suppression. The TLOW determined by John Canny, he said in his paper "A Computational Approach to Edge Detection" that "The ratio of the high to low threshold in the implementation is in the range two or three to one." That means that in terms of this implementation, we should choose TLOW ~= 0.5 or 0.33333.

Follow_edges

This is a subfunction of `apply_hysteresis`, aiming to remove all the possible edges, or we call weak edges. The idea of algorithm is that all the pixel caused by noise but regarded as edge are isolated, which means its neighbors are not edge. It's an easy way using recursive function to go through edge, like go through tree structure. In the later part, I will change this recursive function to iterative function, because synthesis does not support that.

ii. Modeling and simulation in IEEE SystemC

In high-level synthesis language, SystemC is supported. It's a good idea that using SystemC to separate the whole program to different modules. There are some advantages,

1. Easy to design by top-down design flow.
2. Convenient to update and reconfigure the program because of hierarchical structure.
3. Clear to simulate and monitor every module status.

1. The Whole Blueprint of Canny

On the first step, we need to change function in C++ into module in SystemC, and for communication, we will instantiate FIFO-type channels from the SystemC standard library. Specifically, use the regular first-in-first-out primitive channel `sc_fifo<T>` where template parameter T is the type of the data we need to communicate. If C++ does not provide an operator for array assignment, we need to wrap the array into a proper class with overloaded operators.[4]

After that, we grossly separate Canny into three part, like Figure 12, Stim, DUT and Mon. Stim will read the data and send to Plat, Plat module will finish canny work. In the end, Mon will compare the result to images we use canny directly produce in order to verify our model, what's more, Mon will print the information we want. Therefore, Stim is responsible for `read_pgm_image`, Plat for `canny` and Mon for `write_pgm_image`.

Similarly, we continuous dividing the Plat module into three second level modules. Shown in Figure

[4] From slide of Professor R.Doemer.

12, the lines actually are channel connected to port.

Here, we still reserve two modules named DataIn and DataOut, in the simulation, they just pass the images, however it is benefit when refine in communication stage that we could continue using the testbench of the former. Because if without these modules, the DUT port will connected directly to Stim, if with, it can act as a buffle.

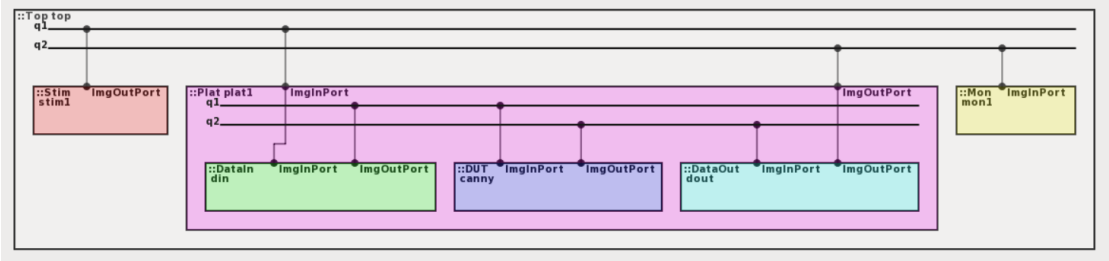

Figure 12

2. Subdivision of DUT Module

On the second step, we will go deep into DUT module and make the local method in Canny encapsulated into individual modules by themselves, like Figure 13.

```
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
```

Figure 13

After completion, we will get Figure 14 and Figure 15, Figure 15 is the final sketch of gaussian smooth, we transfer kernel and wsize from `blux` to `blury`, not `make_gaussian_kernel` directly to `blury`

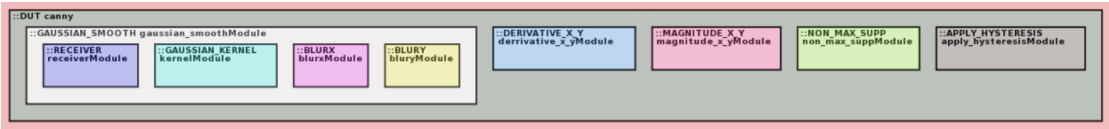

Figure 14

In code style, I put the function of SC_Thread outside the module and use namespace to define the function in order to make the code look clearer. Therefore, in my Canny.cpp, the first half is declaration of module and the last half is definition of function, like as follows,

```
void DERIVATIVE_X_Y::derrivative_x_y(short int *smoothedim, int rows, int cols,
short int *delta_x, short int *delta_y)
```

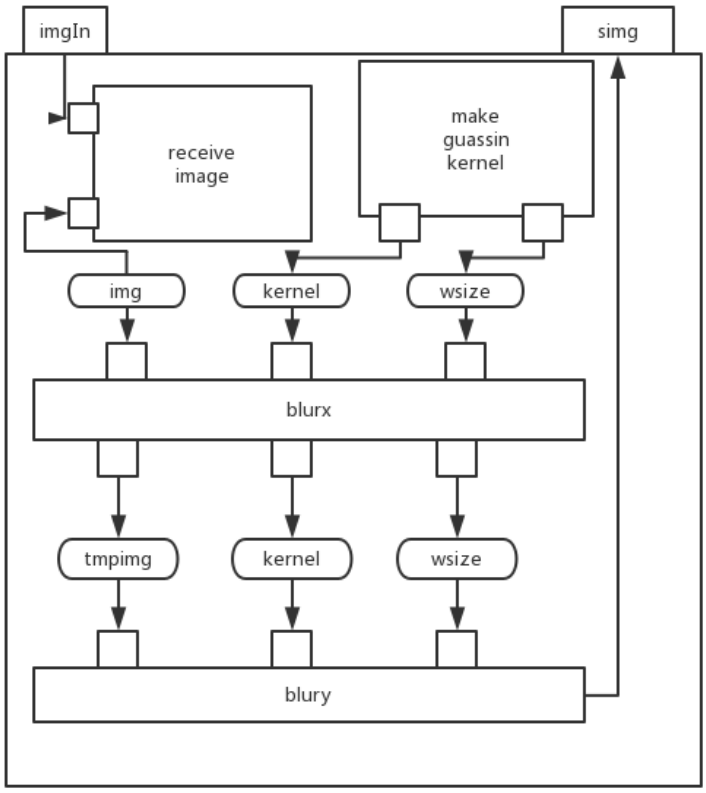

Figure 15

Every function become an individual module, and their ports are connected by channel to communicate.

We write a Makefile to compile and test our result, this script you can find in doc file.

In Makefile, we need to set stack size, by default, the stack has limited size storing in contiguously in memory, and if stack grows too large, it will overwrite the heap. Using `limit` or `limit -h`, we can view soft limits and hard limits in tcsh. We can find soft limits is 10240KB and hard limits is unlimited. Here, we set stack size 128MB in order to avoid segmentation default.

3. Performance estimation and throughput optimization

On the third step, we want to know the performance of our model. Firstly, we execute our program on our server. Before execution, we use -pg option when compiling, then we can use gprof to analyze our model, getting Figure 16

Gaussian_Smooth	42.45%
|------ Gaussian_kernel	0%
|------ BlurX	20.40%
|------ BlurY	22.05%
Derivative_X_Y	5.19%
Magnitude_X_Y	16.17%
Non_Max_Supp	26.46%
Apply_Hystevesis	9.71%

Figure 16

We execute the canny in Raspiberry Pi 3B+, a popular open source device and print every module average real executed time per image, then get Table 1 as follow,

main runs time totally	170.992673 secs
canny runs average time	5.680213 secs
guassian smooth runs average time	3.155554 secs
guassian kernel runs average time	0.000013 secs
blurx runs average time	1.450427 secs
blury runs average time	1.692159 secs
derrivative_x_y runs average time	0.406668 secs
magnitude_x_y runs average time	0.854389 secs
non_max_supp runs average time	0.663524 secs
apply_hysteresis runs average time	0.570007 secs

Table 1

In simulation level, all the work is done at 0 sec, if we want to let our SystemC model to imitate the raspiberry pi execution, we are supposed to put these delays into our SystemC model, then let the SC_Thread wait the same time. Without any delay, all the work will be completed in 0 time, like Figure 17, because now we simulate our model not synthesis and test it on the board.

```
    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.
```

Figure 17

With delays, we get Figure 18,

```
[...]
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.
```

Figure 18

4. Model refinement for pipelining and parallelization

Obviously, 0.549 FPS for a real-time video is quite low performance, we need to refine our model now. As you can see, our module is similar to pipeline structure, so we want let it be true pipeline. In Figure 16, module ` DERIVATIVE_X_Y` outputs the dx and dy for ` MAGNITUDE_X_Y` as well as ` NON_MAX_SUPP`, we need to cancel this hop, let all the data `NON_MAX_SUPP` gets directly from its former.

After that, we will get Figure 19 and sketch map Figure 20

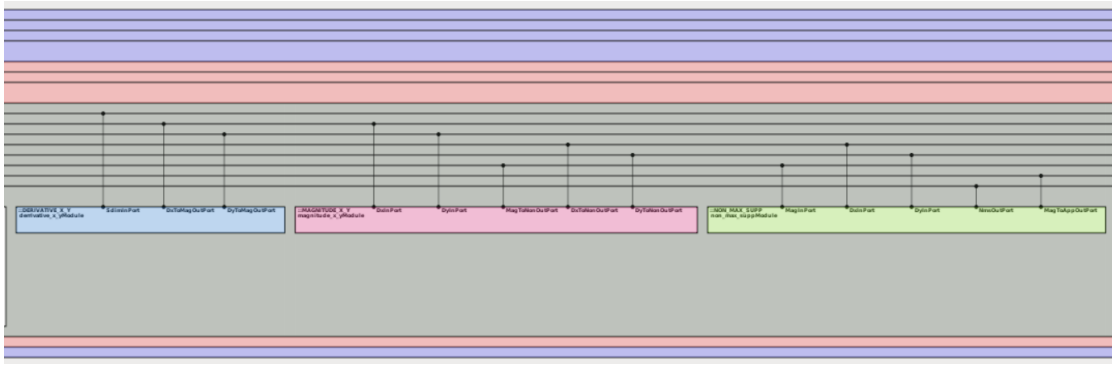

Figure 19

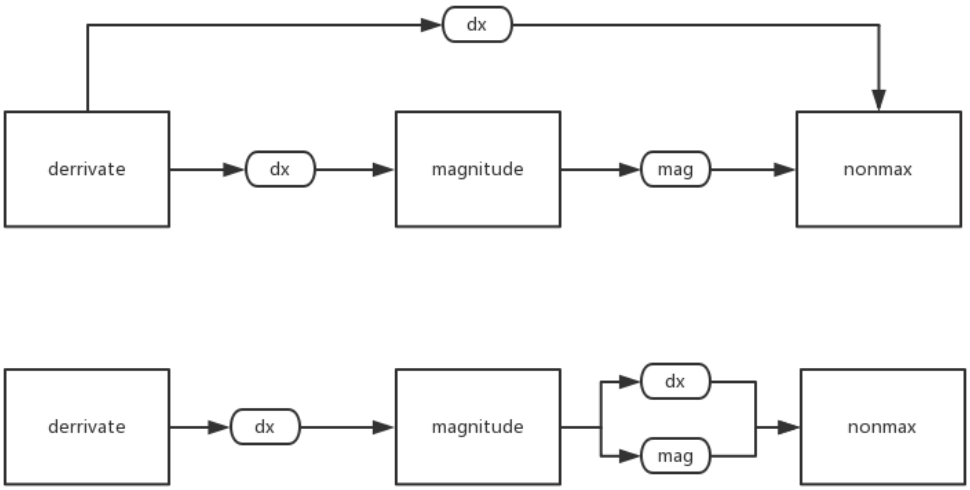

Figure 20

For image processing, we usually deal with pixels row by row, however, we could handle it by dividing into four methods working together. Therefore, we change our `BLUR_X` and `BLUR_Y` into parallelization structure. We use sc_event to notify the `start_blurx` that four subthreads have finished their work and request the next image. Finally, we improve our program throughout from 0.549 FPS to 0.971 FPS, like Figure 21

```
27582 ms: Monitor recerived frame 24 with    4120 ms delay.
27582 ms: 1.030 seconds after previous frame 23, 0.971 FPS.
27000 ms: Stimulation sent frame 28.
28612 ms: Monitor recerived frame 25 with    4120 ms delay.
28612 ms: 1.030 seconds after previous frame 24, 0.971 FPS.
28000 ms: Stimulation sent frame 29.
29642 ms: Monitor recerived frame 26 with    4120 ms delay.
29642 ms: 1.030 seconds after previous frame 25, 0.971 FPS.
29000 ms: Stimulation sent frame 30.
```

Figure 21

The reason why we can parallel the `BLUR_X` and `BLUR_Y` is that every pixel gets it value by values of its neighbors' (calculating by convolution) and go through row by row, like Figure 22. Therefore, we divide the whole image into four parts and handle them concurrently. After processing in X direction, we do the same thing in Y direction.

For pixel Pj, we blur it in X direction,

$$window\ size = 2i + 1,$$

$$P_j = sum([x_{j-i}, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_{j+i}] * [\,gaussain\ kernel])$$

Here * is convolution, not matrix multiple. Arrows mean movement direction.

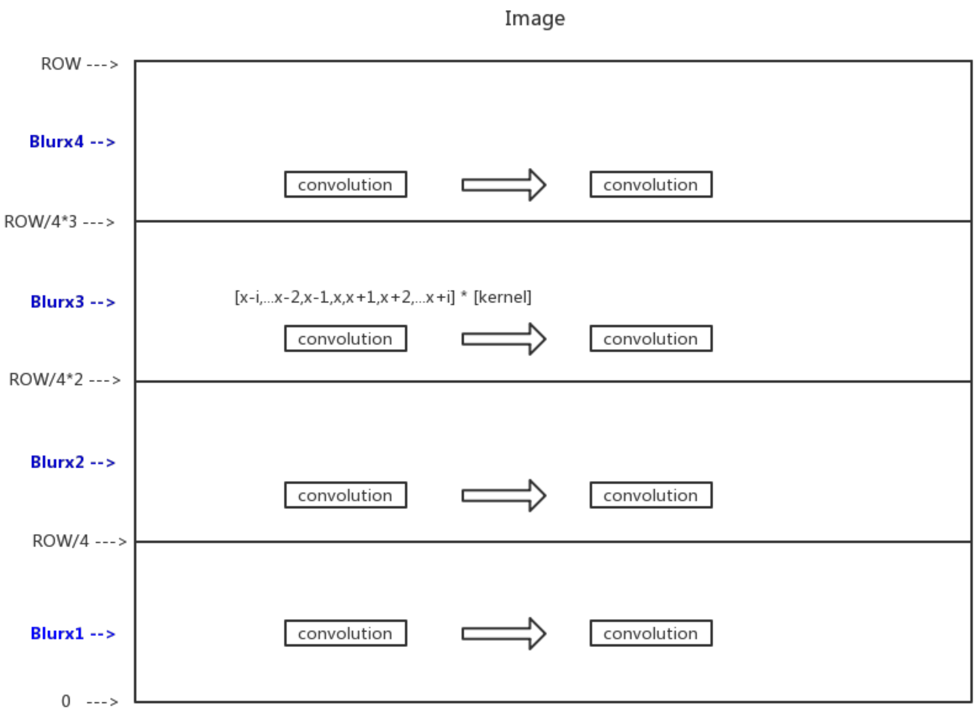

Figure 22

If we want to refine our model deeply, we can process `Derrivative_x_y`, `Magnitude_x_y` and so on, similarly to `Blurx` and `Blury`, because they also deal with the pixel one by one or row by row. According to the number of our multi-cores system, we can divide them into several parts processing parallel.

When test SystemC model again, there is a quiet interesting thing that the simulator run-time does not improve. Obviously, our SystemC model has paralleling in module `BLUR_X` and `BLUR_Y`, but simulator does not have. The simulator is still executed in a single thread, if we can find a way to let every module be executed by one thread in a multi cores system, then we can accelerate simulation time. Another method is that we can use SC_method instead of SC_thread and other effective and efficient data structure in our model.

5. Real-time video performance results

After optimization of pipeline and parallelization, we reduce our `BLUR_X` and `BLUR_Y` time to one forth, from 1710ms to 230ms. Actually, there also are several ways to improve throughout, like using g++ optimization option, we can use -O2 or -O3 when compiling. Using -O2 can save nearly 70% time, that is quite big improvement and -O3 is the best. Also, depending on Raspiberry pi specification, we use -mfloat-abi=hard, -fmpu=neon-fp-armv8, and –mneonfor-64bits option to get amelioration.

Here we have time of execution by the server (Figure 16), SystemC model execution time without optimization (Figure 18), SystemC model execution time after pipeline and palleling (Figure 21) and with -O3 optimization. We will not use time of execution by the server, because that is not execution time of real-time system. Even in Raspiberry Pi, the CPU is ARM architecture but x86 in the server, they have different performance in different types of calculating. Therefore, the result of Assignment 6 just reflects every module occupies percent of execution time generally and roughly.

Considering relative time in the past work, we give up Assignment 6 execution time. After comparing the results, we obtain Table 2 of execution time (ms) and Table 3 of FPS,

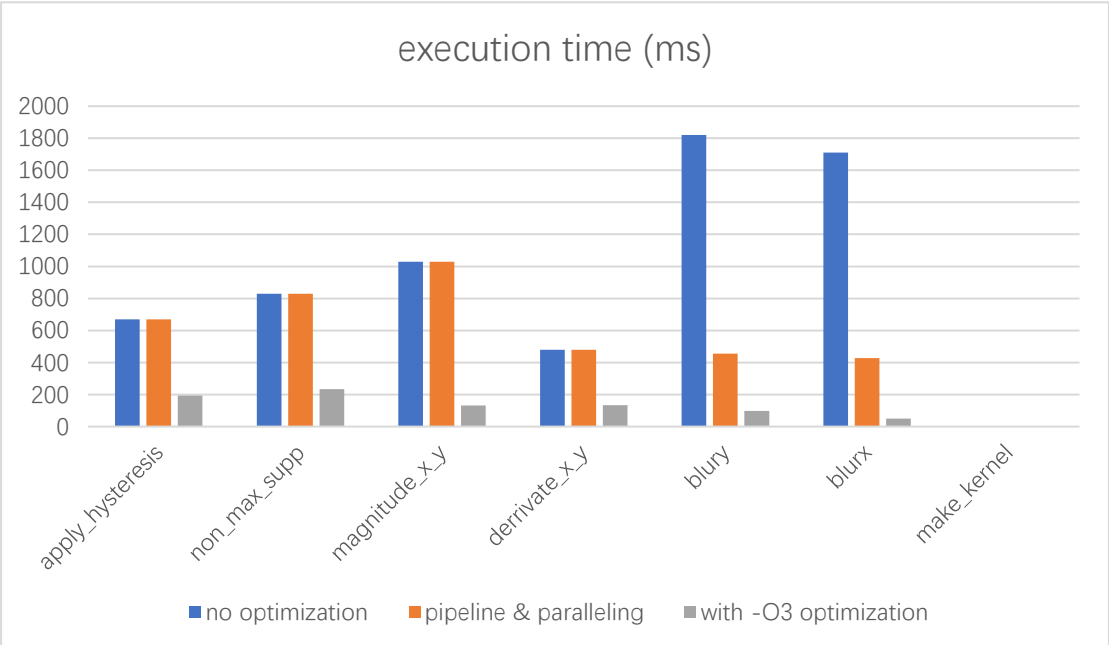

Table 2

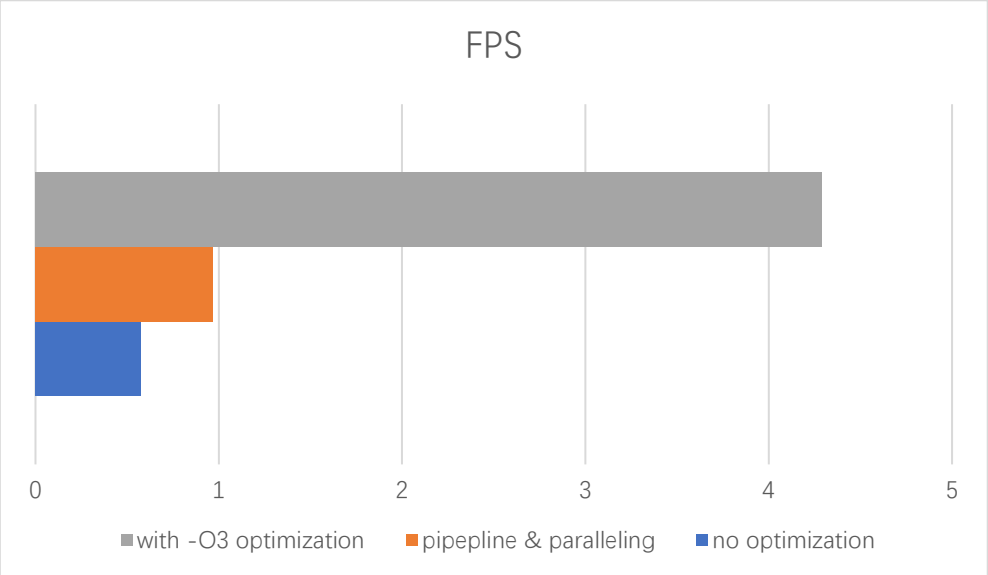

Table 3

On the other hand, using int type instead of float type in `non_max_supp` not only not saves time but also loses the accuracy. On average level, we spend 2 times time and miss serveral pixels in 2704*1520 pixels. I give up this method.

III. Summary and Conclusion

i. Lessons Learning

At the moment I finished this report, I felt relax. “Eh, learning quite a lot”, I am filled with satisfaction.

In this course, we studied how to model and design embedded system, what is systemC and how we can use systemC into simulate our embedded system. In the assignments, we refine our model more deeply step by step to obey top-down design flow. I think there are some rules we need to obey or persist in during development. The first thing is we need a whole picture about what we will do in our embedded system, then we need to design our model separated into several models and figure out how they are connected to each other. That will be useful, if we upgrade or maintain our model on this basement. Also, it is important to optimize our model, we test it and make it clear what is bottleneck, removing the things embedded system may not support and adding the tricks to accelerate speed. Surely, we are supposed to make sure our model validity all the time, and that’s why we always compare the current result to previous results after finishing. Last but not least, enjoying the whole project and be happy to every small progress. That’s what I learnt in this course.

ii. Future work

Other methods of saving time

We still can do lots of work on our model, for example, in order to get deep optimization, we can continue to parallel the module `derrivate_x_y` and `magnitude_x_y`. In `follow_edge`, it is a recursive function, we also can transform it into iterative function to reduce the function call time and save context time. Also, we can use Roberts, Prewitt edge detection operator in finding the intensity gradient of the image instead of Sobel, some can save time. However, I don not think we are supposed to focus on how to improve our algorithm in detail, we can use high performance embedded device and useful architecture in real world to realize high FPS.

1. Architecture

Now I am try to get synthesis on my FPGA board. The plan is that we use drone to capture real-time video, due to limitation of load and power, the drone cannot take a high-performance chip on it, so we have to let drone transfer the data to our edge computing device, like FPGA which is good at high speed concurrent computation.

In further work, I want to introduce the machine learning into my project, we are known that edge detection of the image can be used in distinguish plate number of cars, but if we have a high-performance device, we can use it in real-time video, which could be regarded as consisting lots of image per second. Machine can accelerate this process and find the best value of sigma, thigh and tlow by enormous training sets, which are pre-setted by ourselves. Machine learning part can be dealt with cloud computing. Therefore, the whole structure of my imagination is like Figure 23,

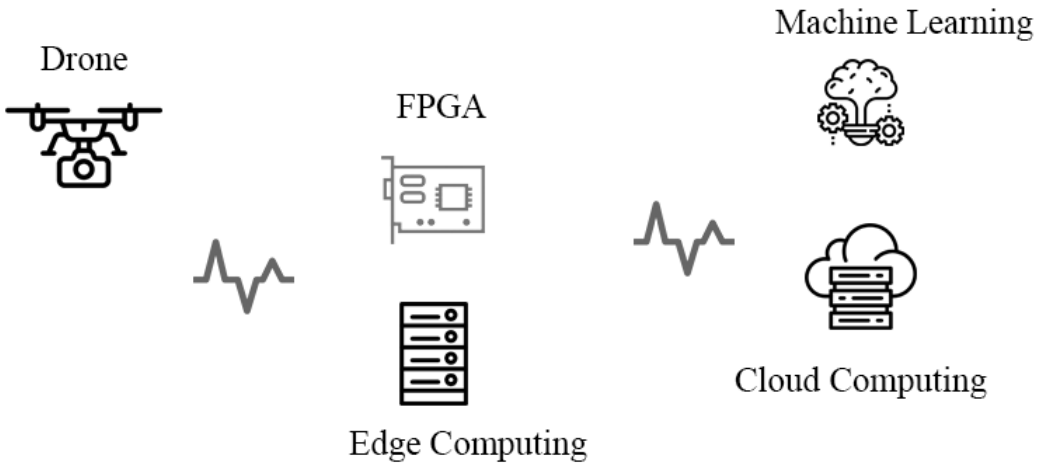

Figure 23

2. Development on FPGA

The device I choose is Pynq-Z2[5], shown in Figure 24

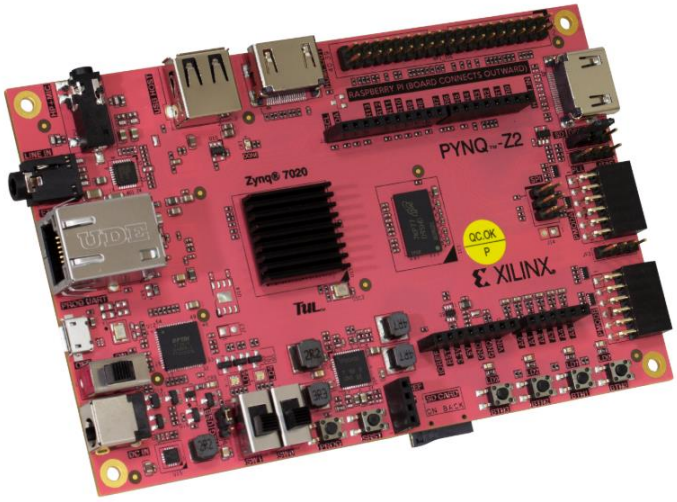

Figure 24

What is PYNQ? (www.pynq.io)[6]

PYNQ is an open-source project from Xilinx that makes it easy to design embedded systems with Xilinx Zynq Systems on Chips (SoCs).

Using the Python language and libraries, designers can exploit the benefits of programmable logic and microprocessors in Zynq to build more capable and exciting embedded systems, like Figure 25.

[5] Detail of Pynq-Z2: http://www.tul.com.tw/ProductsPYNQ-Z2.html

[6] What is PYNQ: http://www.pynq.io/

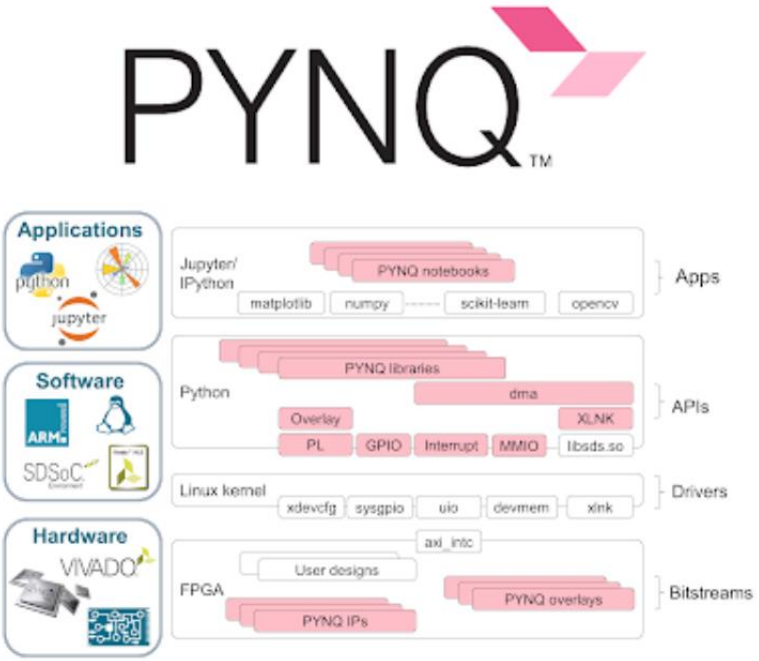

Figure 25 (c) 2018 www.pynq.io

With Pynq, we can generate canny IP and use in python project, combining high-performance of FPGA and plenty of Python library in machine learning.

Pynq overlay is a bridge of them. Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System of the Zynq into the Programmable Logic. Overlays can be used to accelerate a software application, or to customize the hardware platform for a particular application, like Figure 26.

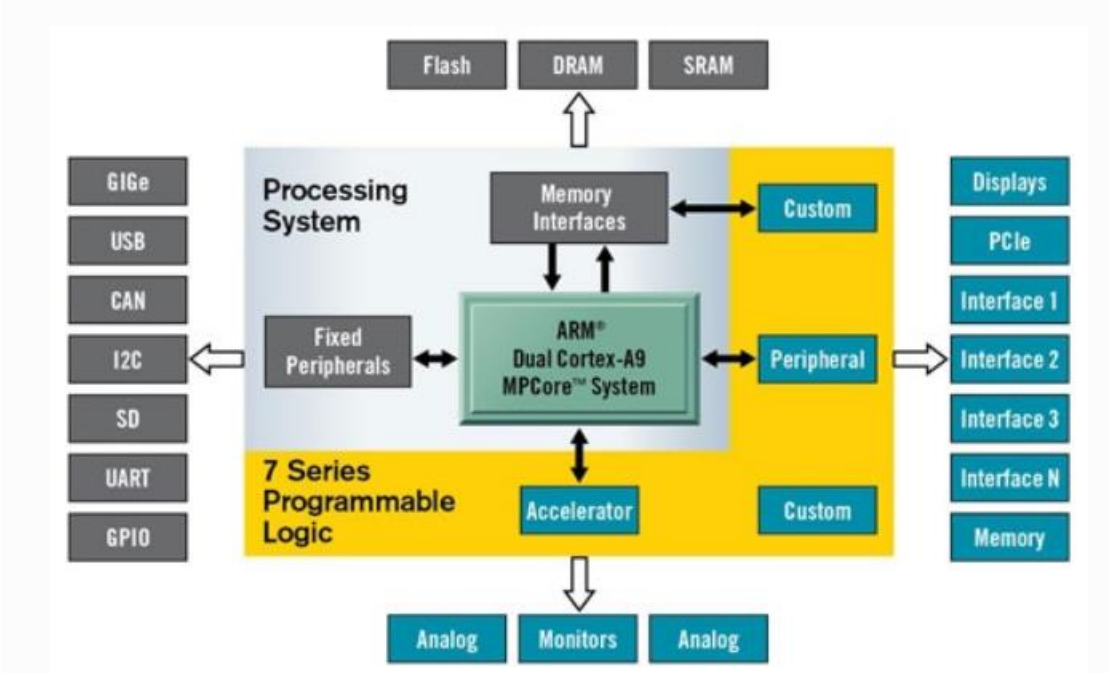

Figure 26 (c) www.pynq.io

What is High-level Synthesis and Vivado HLS?[7]

Vivado HLx is an IDE tool produced by Xilinx. Vivado High-Level Synthesis included as a no cost upgrade in all Vivado HLx Editions, accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx programmable devices without the need to manually create RTL.

3. Applying SystemC model in Vivado HLS

The first step is to divide Top module into three separate module, because the Stim and Mon have operations of reading and writing images, which are not support by HLS, so these modules are more suitable to place in simulation part, or we call testbench. What's more, we separate the canny into several files, put all the declarations in val.h in order to avoid multiple definitions. Then we run C Simulation and get the same result to before.

(Current working...)

The second step is to synthesis, that will cause lots of errors, mainly including some variables like sc_event, recursive function `follow_edge` in Canny and SC_thread not supported.

1. Transferring the recursive function `follow_edge` into iteration

`follow_edge` like tree traversal, we are known that iteration with a stack to store records can replace recursion. Here, I use standard C++ library <stack>, I can pass C Simulation, but I cannot pass C Synthesis, I will use array to achieve stack in the next week. Recursion increase function call time, here I put the code into Raspiberry Pi to have a try, however to my surprise, time assuming increases after converting to iteration.

2. Using SC_method or SC_cthread instead of SC_thread.

In Xilinx HLS, SC_thread is not supported, however, we can use SC_method or SC_cthread. SC_cthread is sensitive to clock, but not to signal. As far as I am concerned, here, in paralleling part of Blurx and Blury, sc_event can be replaced by sc_signal, but not clock, as well as SC_method has higher performance than SC_cthread.

For high performance in communication level, it is not good idea to use individual line to transfer signal between 2 module, that will cause increase the bus we need in the end.

3. Solving the problem self-defining struct cannot be synthesis

We define some IMAGE, SIMAGE and so on self-defining struct, which are not supported by HLS. Considering the communication level, we need to transform the channel to the signal, like Figure 27,

[7] Vivado Design Suite User Guide high-level synthesis: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf

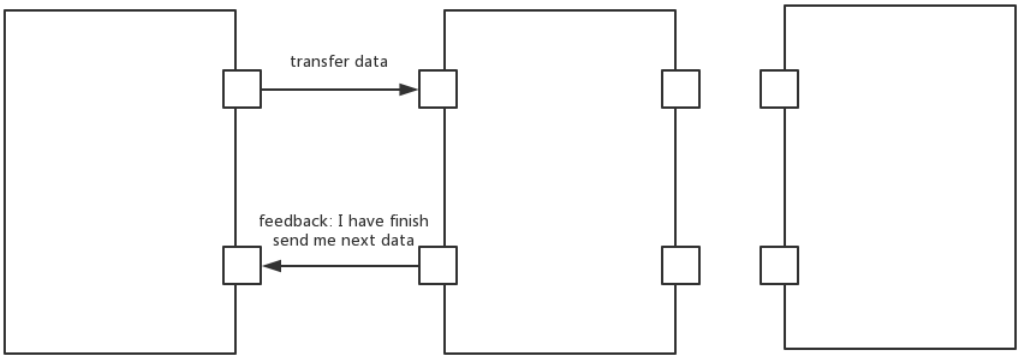

Figure 27

To improve the utilization of the bus, we prefer the follow model, like Figure 28,

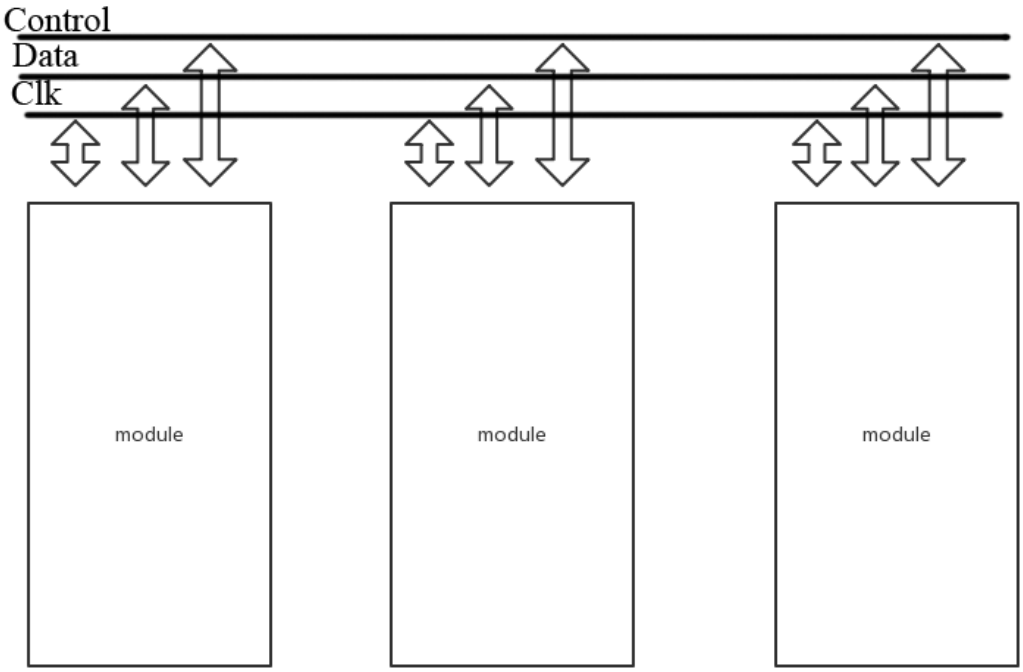

Figure 28

IV. Reference

[1] Wikipedia contributors, "Cyber physical systems," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/wiki/Cyber-physical_system (accessed December 3, 2018)

[2] 2001 University of South Florida, Tampa, *Canny C++ source code*: http://marathon.csee.usf.edu/edge/edge_detection.html (accessed December 9, 2018)

[3] Wikipedia contributors, "Canny edge detector," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/wiki/Canny_edge_detector (accessed December 9, 2018)

[4] Rainer Dömer. *Embedded Systems Modeling and Design* [Slides]. Retrieved from Center for Embedded and Cyber-physical Systems, University of California, Irvine Website: https://eee.uci.edu/18f/16905/assignment.html

[5] Tul. *Detail of Pynq-Z2.* Retrieved from http://www.tul.com.tw/ProductsPYNQ-Z2.html

[6] Pynq. *What is Pynq.* Retrieved from http://www.pynq.io/

[7] Xilinx, *Vivado Design Suite User Guide high-level synthesis* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf