Modeling of a Canny Edge Detector for Embedded Systems Design

Kartik Kaul

12/03/2018

ECPS 203 (Embedded Sys Modeling and Design)

Abstract

Much of our knowledge of computing algorithms and their applications is restricted to PCs, laptops, mainframe servers and other such standalone environments. The disadvantage of working on such systems is that we are never able to understand the efficiency of algorithms and applications when the resources available to run the program are almost infinite. In real life scenarios we are always limited by time and resources and hence must learn to accommodate the same within our design flow so as to build robust softwares.

Even though we are surrounded by embedded systems we seldom understand their intricacies and try to ask ourselves how programs run on an embedded or hardware level. Even in the computing world, for a long time hardware and software have been separate areas but recently with rise of IoT and related distributed system applications, we have seen the divide being bridged.

In this report we discuss a robust example of how an existing image processing algorithm such as Canny can be modelled on to a system level design using systemC and be used effectively for many applications, specifically to generate a structural overview of buildings when mounted on an embedded system such as an RPi along with a Camera to be used to record a live video stream upon a Drone in real time. Such a design could be useful in several areas especially in the field of civil engineering, in structural health monitoring which is essential in Disaster Management and Prevention, in stress testing of structures such as Bridges, Dams and Highways before unveiling to the public, and many more.

However, the Register Transfer Logic (RTL) design associated with any hardware implementation poses quite a challenge for any designer. Each assignment has to be timed, and be as resource specific as possible. Which registers to use, how to access the ALU, to have SIMD or VLIW, how to access the memory? Thus programming in High Level Languages such as C/C++ has always been preferred and the subsequent machine code generation is handled by complex compilers and interpreters.

Thus, to mount a high level design application on a simple embedded system board, we will need to refine our model so that the application is able to abstract some aspects of RTL beforehand. Thus we use modelling languages such as systemC which combine the simplicity of High Level Design and Programming Logic with time and memory constraints imposed by RTL design. Such languages also give us an insight into multithreading and pipeline based programming which comes handy while resolving the constraints mentioned above.

Finally at the end of the report, we discuss the conclusion of the work done in achieving our goal, and future improvements which can be incorporated in this area.

Introduction

Embedded system modeling and design concepts:

Driven by ever-increasing market demands for new applications and by technological advances that allow designers to put complete many-processor systems on a single chip (MPSoCs), system complexities are growing at an almost exponential rate. Together with the challenges inherent in the embedded-system design process with its very tight constraints and market pressures, not the least of which is reliability, we are finding that traditional design methods, in which systems are designed directly at the low hardware or software levels, are fast becoming infeasible. This leads us to the well-known productivity gap generated by the disparity between the rapid paces at which design complexity has increased in comparison to that of design productivity.

One of the commonly-accepted solutions for closing the productivity gap as proposed by all major semiconductor roadmaps is to raise the level of abstraction in the design process. In order to achieve the acceptable productivity gains and to bridge the semantic gap between higher abstraction levels and low-level implementations, the goal now is to automate the system-design process as much as possible. We must apply design-automation techniques for modeling, simulation, synthesis, and verification to the system-design process. However, automation is not easy if a system-abstraction level is not well-defined, if components on any particular abstraction level are not well-known, if system-design languages do not have clear semantics, or if the design rules and modeling styles are not clear and simple.

On the modeling and simulation side, several approaches exist for the virtual prototyping of complete systems. These approaches are typically based on some variant of C-based description, such as C-based System-Level Design Languages (SLDLs) like SystemC or SpecC. These virtual prototypes can be assembled at various levels of detail and abstraction. The most common approach in the system design of a many-processor platform is to perform co-simulation of software (SW) and hardware (HW) components. In algorithmic-level approaches in designing MPSoCs, we use domain specific application modeling, which is based on more formalized models of computation, such as process networks or process state machines. These modeling approaches are often supported by graphical capture of models in terms of block diagrams, which hide the details of any underlying internal language. On the other hand, the code can be generated in a specific target language such as C by model-based-design tools from such graphical input.

The IEEE SystemC language:

SystemC is a C++ class library and a methodology that you can use to effectively create a cycle-accurate model of software algorithms, hardware architecture, and interfaces of your SoC (System On a Chip) and system-level designs. You can use SystemC and standard C++ development tools to create a system-level model, quickly simulate to validate and optimize the design, explore various algorithms, and provide the hardware and software development team with an executable specification of the system. An executable specification is essentially a C++ program that exhibits the same behavior as the system when executed. C or C++ are the language choice for software algorithm and interface specifications because they provide the control and data abstractions necessary to develop compact and efficient

system descriptions. Most designers are familiar with these languages and the large number of development tools associated with them.

The SystemC Class Library provides the necessary constructs to model system architecture including hardware timing, concurrency, and reactive behavior that are missing in standard C++. Adding these constructs to C would require proprietary extensions to the language, which is not an acceptable solution for the industry. The C++ object-oriented programming language provides the ability to extend the language through classes, without adding new syntactic constructs. SystemC provides these necessary classes and allows designers to continue to use the familiar C++ language and development tools. If you are familiar with the C++ programming language, you can learn to program with SystemC by understanding the additional semantics introduced by the SystemC classes; no additional syntax has to be learned. If you are one of the many that are more familiar with the C programming language, you need to learn some C++ syntax in addition to the semantics introduced by the classes. The use of C++ has been kept to a minimum in SystemC. If you are familiar with the Verilog and VHDL hardware description languages and the C programming language, learning SystemC will be easy.

SystemC supports hardware-software co-design and the description of the architecture of complex systems consisting of both hardware and software components. It supports the description of hardware, software, and interfaces in a C++ environment.

The following features of SystemC version 2.0 allow it to be used as a codesign language:

- Modules: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
- Processes: Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers.
- Ports: Modules have ports through which they connect to other modules. SystemC supports single-direction and bidirectional ports.
- Signals: SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can have only one driver.
- Rich set of port and signal types: To support modeling at different levels of abstraction, from the functional to the RTL, SystemC supports a rich set of port and signal types. This is different than languages like Verilog that only support bits and bit-vectors as port and signal types. SystemC supports both two-valued and four-valued signal types.
- Rich set of data types: SystemC has a rich set of data types to support multiple design domains and abstraction levels. The fixed precision data types allow for fast simulation, the arbitrary precision types can be used for computations with large numbers, and the fixed-point data types can be used for DSP applications. SystemC supports both two-valued and four-valued data types. There are no size limitations for arbitrary precision SystemC types.
- Clocks: SystemC has the notion of clocks (as special signals). Clocks are the timekeepers of the system during simulation. Multiple clocks, with arbitrary phase relationship, are supported.
- Multiple abstraction levels: SystemC supports untimed models at different levels of abstraction, ranging from high-level functional models to detailed clock cycle accurate RTL models. It supports iterative refinement of high level models into lower levels of abstraction.

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

Structure of the Canny edge detection algorithm:

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. Canny also produced a computational theory of edge detection explaining why the technique works.

The technique is used to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations. The general criteria for edge detection include:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible
2. The edge point detected from the operator should accurately localize on the center of the edge.
3. A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

To satisfy these requirements Canny used the calculus of variations – a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but it can be approximated by the first derivative of a Gaussian.

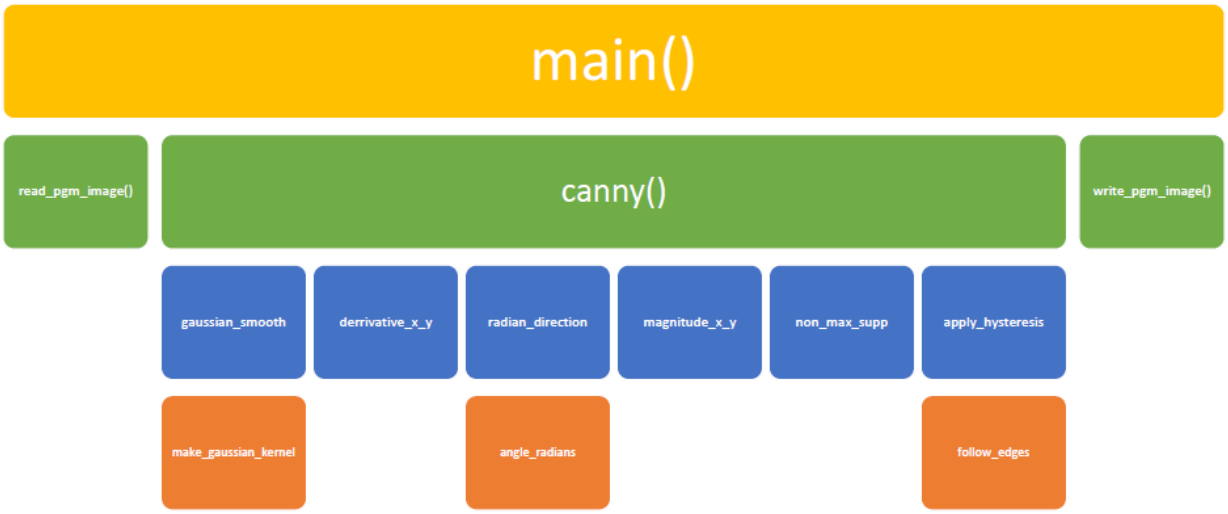
Among the edge detection methods developed so far, Canny edge detection algorithm is one of the most strictly defined methods that provides good and reliable detection. Owing to its optimality to meet with the three criteria for edge detection and the simplicity of process for implementation, it became one of the most popular algorithms for edge detection.

The Process of Canny edge detection algorithm can be broken down to 5 different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

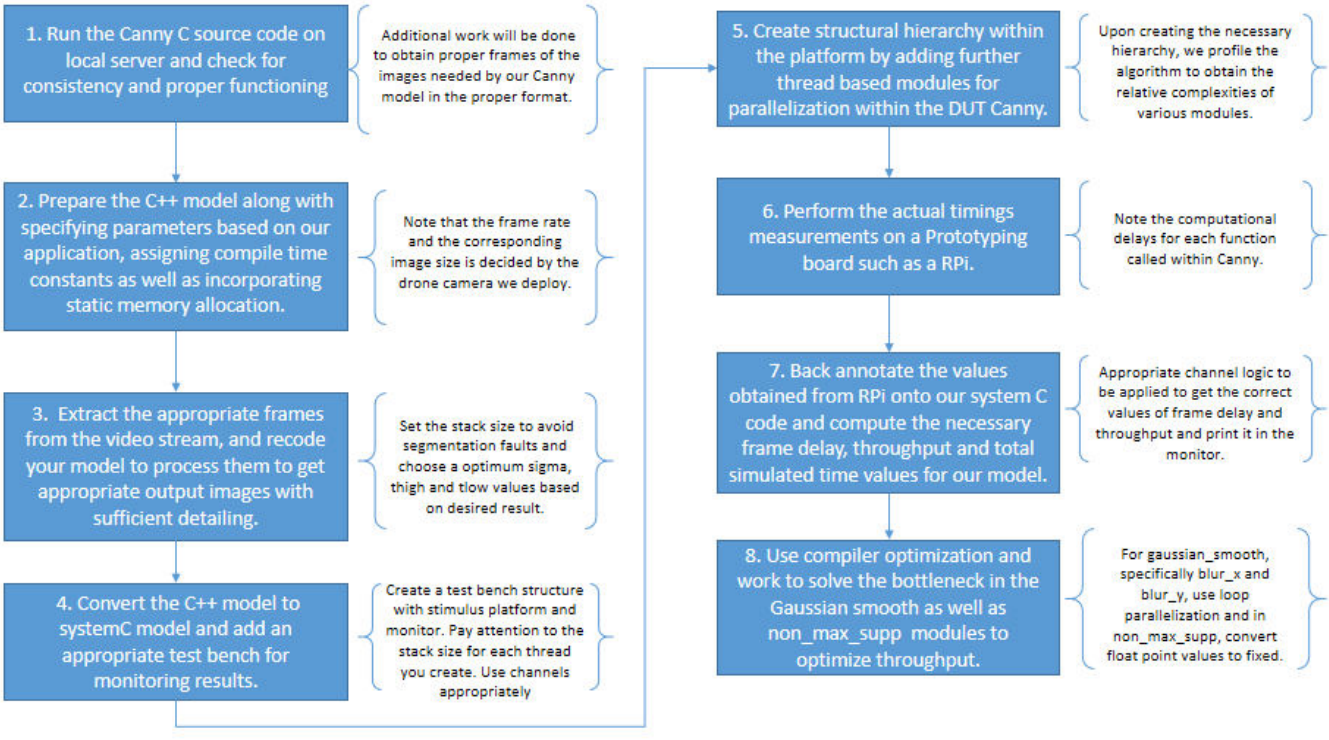
Modeling and simulation in IEEE SystemC:

The functional flow of our original Canny model looks somewhat like this diagram below:



Our objective is to adjust it for running on an embedded system to be mounted on a drone. Hence, step wise we need to convert the model into a compatible systemC model which not only runs in a time bound fashion but also considers the memory and runtime constraints. For example, one such refinement is to get rid of dynamic memory. Dynamic memory allocation (i.e. the use of functions malloc(), calloc(), and free()) is clearly not feasible in a hardware implementation, because the desired SoC cannot instantiate a new memory chip at runtime! Thus, we will use static arrays with fixed sizes at compile time.

Based on many such tasks, I have come up with the following flow diagram for converting the algorithm from a C compatible model to a systemC model.

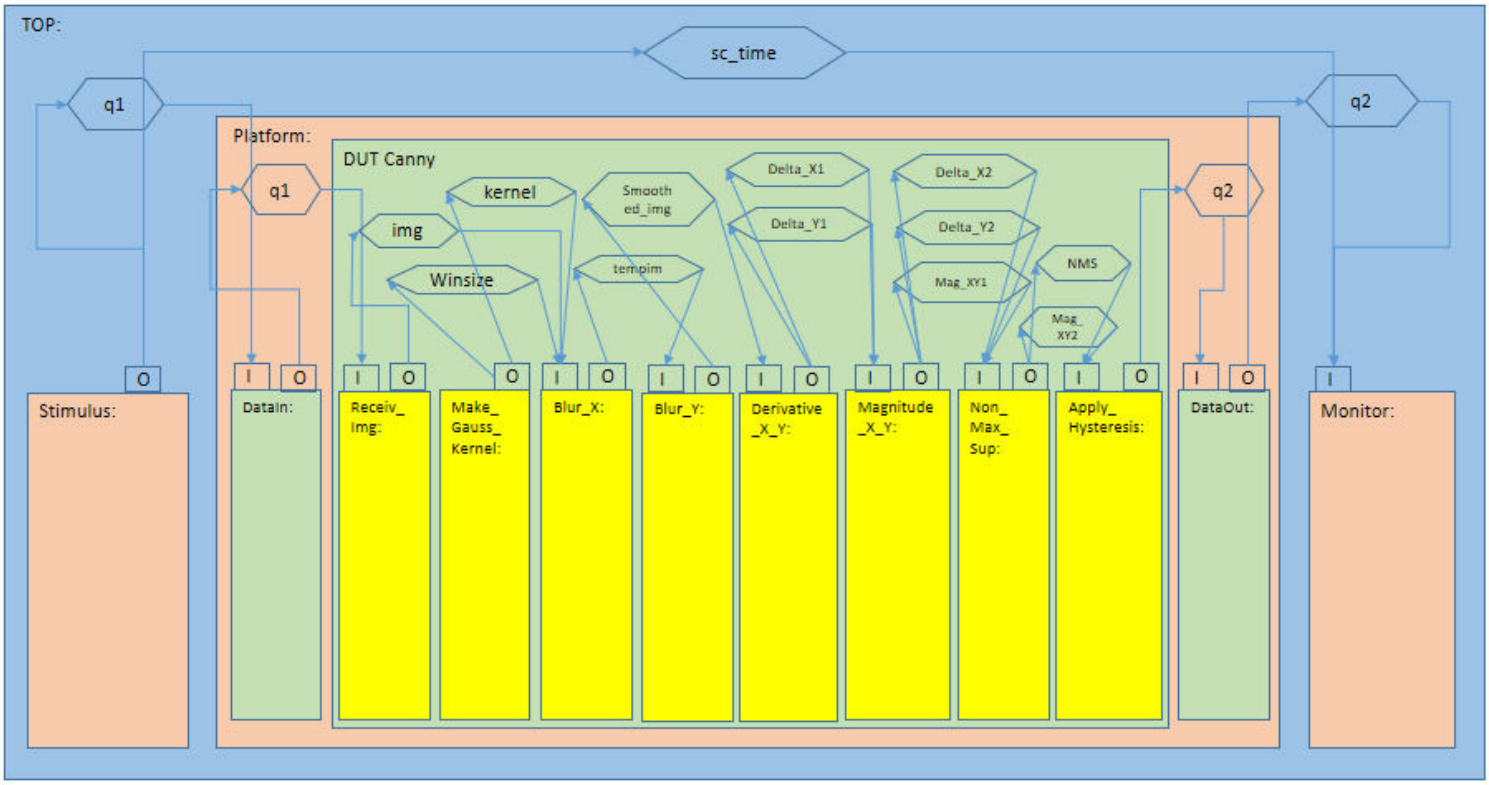


Another crucial task we must consider is the stack size allotment. The higher image resolution (as required by our drone video application and the image resolution of the drone camera) naturally leads to higher memory usage of our application. Specifically, the array variables holding the image data grow large. Note that many of those variables are local variables which get allocated on the stack. At the same time, the stack size of a process in the Linux environment is typically limited. If so, your application will "crash" with a segmentation fault or similar memory error. To avoid stack overflow, you can adjust the stack space allocation in your Linux shell.

Specifically we set the size of the stack at 128 MB which is sufficient.

Model refinement for pipelining and parallelization:

Our aim is to eventually use parallization and pipelining to enable a multithread functionality so that we have maximum efficiency using least resources available (as is with embedded systems). The following is the final pipelined model we obtain after performing all the steps given above. The TOP is the main module containing 3 sub modules namely, *Stimulus, Platform and Monitor.* Platform gets further divided into sub modules namely *DataIn, DUT and DataOut.* In the Platform module, the DataIn module should, in an endless loop, receive an input image and pass it unmodified to the DUT. Similar, the DataOut module should, also in an endless loop, receive an input image from the DUT and pass it on. These two instances will be needed later during model refinement. They will allow our test bench to remain unmodified even when later in the design flow the communication to the DUT is implemented via detailed bus protocols.



Performance estimation and throughput optimization:

After we obtain our parallized model we need to get an estimation of the time so as to measure our performance. The first step towards the same is obtained via using the profiling tool to get a rough image even though one must note that the ***profiling*** tool can give varying results based on hardware specifications of the host machine as well as the iteration it considers as an optimum estimate, which again will vary every time we run it.

```
Gaussian_Smooth                                 38.6%
|------ Receiv_Img                      0.0%
|------ Make_Gaussian_Kernel    0.0%
|------ Blur_X                          17.4%
|------ Blur_Y                          21.1%
Derivative_X_Y                          7.4%
Magnitude_X_Y                           16.5%
Non_Max_Supp                            25.7%
Apply_Hysteresis                        11.8%
                                        Total:  100%
```

But still, we can deduce clearly that the issue is the Gaussian Smooth Block, specifically the Blur_X and Blur_Y modules as well as the non_max_supp module. These are the bottlenecks which we shall refine as we proceed further. Next we do some real time estimation using a prototype board such as a raspberrypi and obtain the accurate run time for each function of the Canny block in the original code.

```
Gaussian_Smooth                         3.53
|------ Receiv_Img                     0
|------ Make_Gaussian_Kernel           0
|------ Blur_X                         1.71
|------ Blur_Y                         1.82
Derivative_X_Y                          0.48
Magnitude_X_Y                           1.03
Non_Max_Supp                            0.83
Apply_Hysteresis                        0.67
Total:                                  6.47
```

After back annotating the above values (given in secs) in our systemC code, we try to measure total simulated time by logging time to the console each time a frame is sent from the stimulus and received at the monitor, along with the frame delay between as a frame starts from stimulus and ends up at the monitor as well as the throughput which is basically the rate at which the frames are received at the monitor per second. This number is essentially the inverse of the frame delay between consecutive frames at the monitor. Thus in Assignment 8 we obtain the following table:

Model	Frame Delay	Throughput	Total simulated time
CannyA8_step1	0 ms	NA FPS	0 ms
CannyA8_step2	17680 ms	NA FPS	59320 ms
CannyA8_step3	17680 ms	0.549 FPS	59320 ms
CannyA8_step4	17680 ms	0.549 FPS	59320 ms
CannyA8_step5	14462 ms	0.971 FPS	33763 ms

The First step makes no time stamp since values have not been back annotated and systemC clock thus records no time delay in simulation.

One thing of note here is that even after pipelining the load to a single stage shift instead of skipping stages via separate channels (step 3 to 4), we obtain no change in the simulated time. Essentially our simulator is sequential by nature as consecutive pipeline stages wait for data to arrive from previous stage via the appropriate channel. The Channel size is also set to 1 which entails only on item to be passed so the stage has to wait for the channel to be emptied before it can put another item on it.

From step 4 to step 5 as seen above, we see a jump in throughput as we try to optimize the Blur_X and Blur_Y modules by using loop parallelization. Essentially we know both the modules operate on the image which is a big 1D array and each accesses the image row wise and column wise sequentially changing the pixel intensity based on a Gaussian distribution. We can parallelize this method so that the multiple sections of the image get blurred simultaneously, so that we save time and hence improve performance.

One thing to note here is that as we create multiple threads which run parallely with our main thread as in case with loop parallelized blur_x and blur_y, we must set stack separately for each thread since we face segmentation fault otherwise.

Next in our final assignment as we attempt to maximize throughput further, we realize that compiler optimization can achieve a much higher performance and hence as we add the optimization as well as CPU flags *-O3 -mcpu=cortex-a53 -mfloat-abi=hard -mfpu=neon-fp-armv8 -mneon-for-64bits -mtune=cortex-a53* and calculate the prototype timings again.

```
Gaussian_smooth                          673 ms
        make_gauss_kernel                  0 ms
        blur_x                           236 ms
        blur_y                           437 ms

Derivative_X_Y                           145 ms
Magnitude_X_Y                            151 ms
Non_Max_Sup                              274 ms
Apply_Hysteresis                         215 ms
Total                                   1458 ms
```

As we back annotate these in our systemC code, we obtain the final performance estimation (step1):

Model	Frame Delay	Throughput	Total simulated time
CannyA9_step1	4599 m	3.65 FPS	8899 ms
CannyA9_step2	4472 ms	3.80 FPS	8706 ms

Step2 is for when we do another optimization technique which can be optional based on every one's own timing result. The fixed point arithmetic option is a tradeoff so to speak, since it can help us reduce the time it takes to run the non_max_supp function (fixed point arithmetic take lesser runtime and memory than floating point arithmetic due to whole values and no overhead dangling bit values to handle) and hence in our systemC canny model, help improve throughput by reducing the interframe delay which by A9 depending on non_max_supp time since it is the slowest stage in the pipeline.

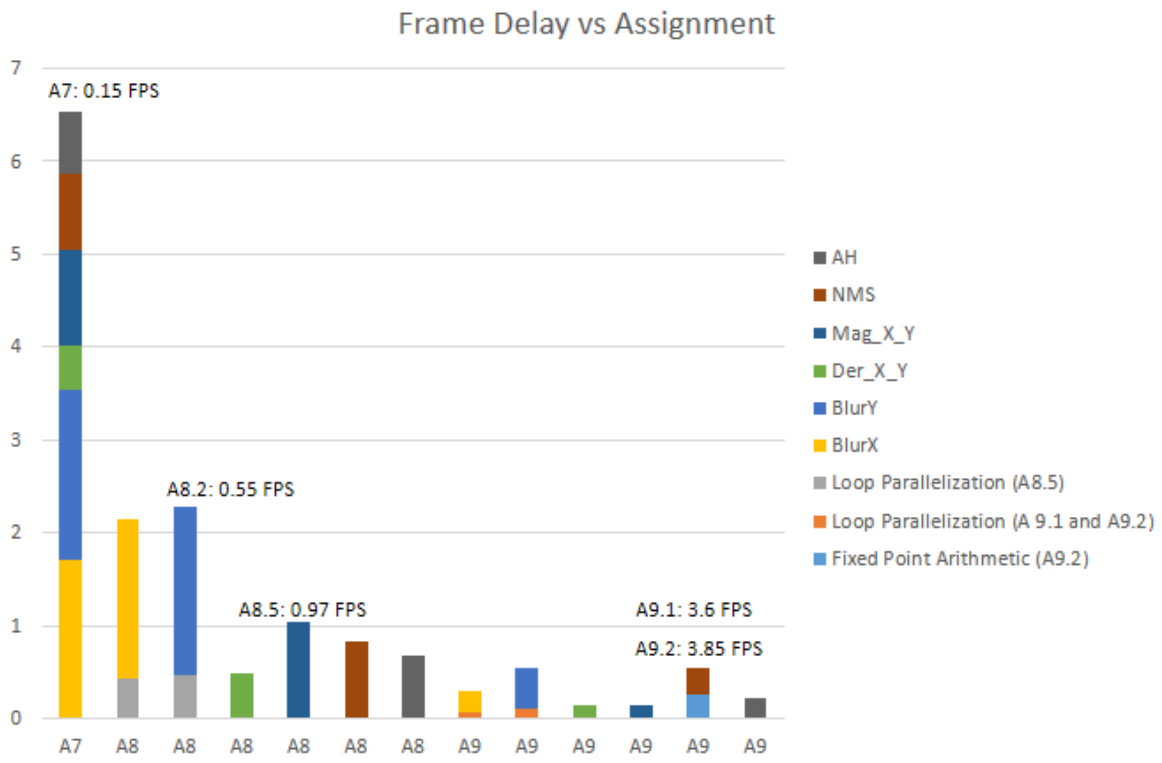
However using the fixed point values instead of floating point ones reduces the accuracy of our output image since we are now using approximate values instead of actual pixel intensity calculations, which is also why our makefile gives error since our output images differ from the ones we get via actual calculations. Therefore we use the Imagediff tool to see how much difference we have based on this approximation and in my case the images varied by approximately 0.1% of the pixels which is tolerable. It give a relatively small jump in throughput of about 0.2 FPS.

Real-time video performance results:

Now, our final throughput value stands at roughly 4 FPS. Our initial target value was 30 FPS and we need to understand outside the scope of the assignments how further we can improve this or can we bring down the target throughput to a more agreeable or even achievable value. Our end user need only view the stream at a coherent rate (15 FPS is sufficient for this, which is also the rate at which most producers stream videos to save bandwidth).

One thing we can do which is worthwhile is reducing the resolution of the images we are processing in our canny example. Our fixed ROW and COLS macros are set at 2704 and 1520 pixels respectively. We can try a small image size say half of this and see how much throughput jump we can obtain.

Also, we can use additional hardware components when we obtain the final Bus Functionality Model (BFM). Our deployment board or mounting embedded system can have a multicore processor with a separate GPU or FPU which are common on SOC designs these days.



Summary and Conclusion

Lessons Learned:

In this course we learned the significance of embedded systems and the various roles they play in creating real life applications such as the one talked about in this report. Not only did we get an introduction into embedded systems concepts and learn SystemC programming but we also picked up several skillsets honed working on various aspects of this project such as working with Linux bash scripts especially on server environments where access is limited and deployment must be efficient.

Moreover, apart from the application we worked on and the infinite possibilities of real life scenarios such as structural health monitoring where it can be used, we also learned about other topics out of scope of the assignments such as the predecessor to systemC, called specC and how it is more robust as it offers many other functionalities.

We learned about various stages of System Design Flow and how as we move from one model to the other along the flow, the level of abstraction reduces and the timing accuracy increases. In an example, we saw how the GSM vocoder application modeling which has much more intricacy in design and complexity can be automated using the SCE tool. It automates the production of various levels of the System Design Flow based on specific parameters and also at one level allows us to even choose the hardware specification we wish to deploy our model with.

Future Work:

Many aspects of the refinement which we did for our canny example took place outside the purview of the source code. Currently we have an application which takes greyscale images of a fixed resolution and generate the output images, but in the future we'd like a fully sustainable application taking in the video stream as a whole, and in color generating an output video stream for the viewer to behold.

There are many other aspects which we could have tested our systemC code on, especially in BFM mode so as to see whether we can maximize throughput using additional hardware extensions such as multicore GPU enabled SOC designs. What other functionalities can be achieved using such hardware additions and how do they feature within our costs is also an aspect to be considered in the future.

Since Canny is quite an old image processing model, I'd be curious as to which other modern image processing algorithms or applications we could use and see parallelized and pipelined efficiently giving better throughput for video streaming. Many such algorithms use ML and NLP libraries and synchronize with the cloud frequently for data analysis. Could such a functionality be supported while running on an embedded system mounted on a drone? All such questions and many more could be well included the purview of this course in the future.

References

- Embedded System Design (*Modelling Synthesis and Verification*) by Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer and Gunar Schirner.
- System C Version 2.0 User Guide www.systemc.org
- Canny Edge detector - https://en.wikipedia.org/wiki/Canny_edge_detector
- The Definitive Guide to SystemC Language, by David C Black, Doulos (Design Automation Conference).