SENG 440 COURSE PROJECT

RGB to YCbCr Color Space Conversion

University of Victoria
Department of Computer Science

James Ryan
V00830984
RyanJA@uvic.ca

James Ryan

Contents

Table of Figures

Introduction

This report outlines my project which was conducted for UVic Seng 440: Embedded Systems. The project topic that I had chosen was on Color Space Conversion, RGB to YCC. I had chosen this topic because I had interest in understanding color spaces, how they related to one another and how they are used within computer systems to generate color information to be displayed to a user. Addition, I felt my skillset was best suited for the tasks required even though I did lack some knowledge in image processing.

This lack of knowledge and experience did end up proving to be a challenge in the project, which resulted in a shift of approach and project specifications in the middle of it. The original intent of this project was to read a JPEG image, do the required conversions, and then write back to an output JPEG file. Originally the JPEG image format was chosen because it was the only one found in my research which both supported RGB and YCC color spaces. However, after numerous hours spent researching and prototyping, the ability to read and write a JPEG file from scratch proved impossible with my understanding of image processing. An example of such is Huffman encoding and decoding, which happened to be another project topic, was required to read and write JPEG formatted images. Further research into the topic revealed that it's recommended to use multi-thousand-line open source libraries to achieve such a task. I made the decision that this was outside the scope of this project and I would not have been able to realistically achieve my original intent, and therefore re-adjusted my project specifications and requirements to the current form outlined in a subsequent section.

Theoretical Background

For a computer to be able to display colors to the user, there is a need for a way to represent the colors which can be stored and used by the computer to display the intended color to the user. A representation of colors is known as a color space. A human can differentiate between colors using receptors in the eyes which respond to Red, Green, and Blue wavelengths and the intensity of the light. The intensity of light is known as the luminance.

A common color space used is called RGB which is used in displays . RGB stores values of Red, Green and Blue using an N-bit integer for each value. The number of bits, N, used to represent each color is dependent on the accuracy needed. The most common number of bits for each color is 8, also known as 24-bit RGB. The RGB signals are not efficient for transmission or storage, due to redundancy. This is where the YCbCr color space comes in. YCbCr represents brightness, or luminance (Y) and two color signals. These color signals are Cb which represents blue minus luminance and Cr which represents red minus luminance. The Cb and Cr signals can

then be compressed in various ways to save space while retaining the original luminance value. This resulting compressed version will not vary significantly from the uncompressed signals because the human eye is more reactive to luminance than the color signals.

Project Specifications and Design

The objective of this project was to convert an image in the RGB color space into the YCbCr color space and compress the image. The resulting image information could then in theory could be transferred to another device over the network, where it is then saved, manipulated or displayed to a user. Therefore, this project it was necessary to perform the following for the main purpose of this project:

- Read an image;
- Convert each pixel into the YCbCr color space;
- Down sample the Cr and Cb values using the average of 2 by 2 pixels for each component;
- After a working solution exists, optimize using various applicable techniques.

At this point, the resulting pixel information could be stored or transmitted for use by another component where it is needed in the overall project.

It was also necessary for testing, such as comparing input and output images, to do the following:

- Convert each result pixel back into RGB;
- Write the image information back into an image that is the same filetype as the original.

However, it was not required to optimize the portion of the project which facilitated testing.

After planning out what needed to be done for this project, I took up researching how to do it. This included brushing up on C because I hadn't worked with it for a couple years, reading the JPEG standard and then the Portable Pixel Map (PPM) standard, and researching color spaces. The next step was to build a functionally correct program.

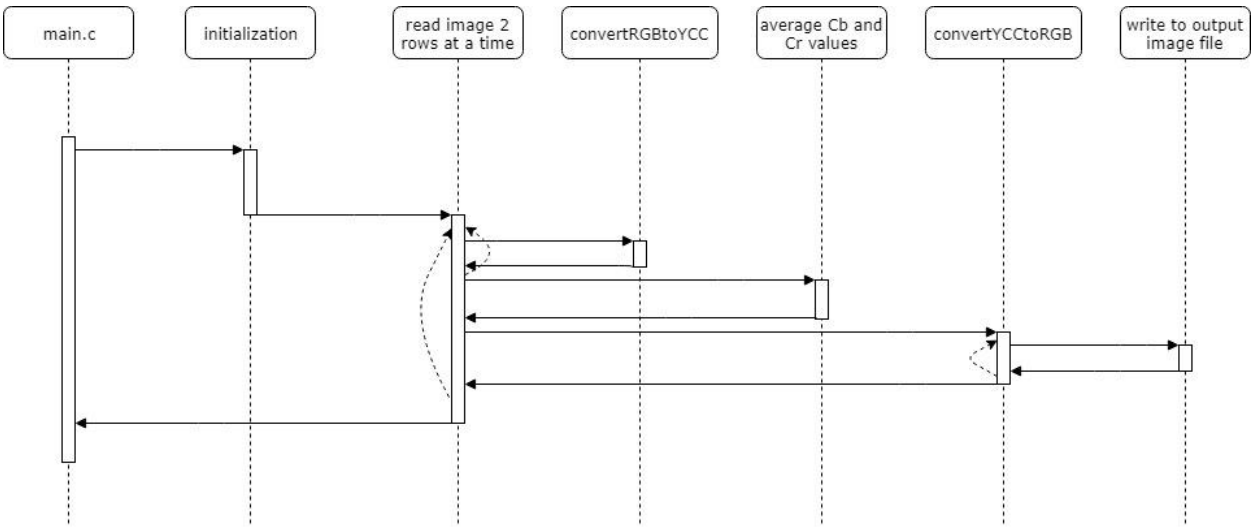

Figure 1: UML Sequence Diagram

Figure 1 shown above shows the sequence of the software produced for this project. What is shown after averaging the Cb and Cr values only exist for the purpose of testing. As mentioned above, it therefore was not optimized.

After building a successful program, I then worked on optimizing the C code.

Optimizations Made to C Code

The first optimization that was made, was to improve code which converted the RGB values to YCbCr. It was obvious that this would be something that had a significant impact on performance as even without profiling the C code, I knew that this set of code was run the most. This was because it had to be done for every single pixel in the input image.

```
uint8_t *convertRGBtoYCC(uint8_t red, uint8_t green, uint8_t blue) {
    uint8_t *YCC = malloc(sizeof(int)*3);
    YCC[0] = (0.257 * red) + (0.504 * green) + (0.098 * blue) + 16;
    YCC[1] = (-0.148 * red) - (0.291 * green) + (0.439 * blue) + 128;
    YCC[2] = (0.439 * red) - (0.369 * green) - (0.071 * blue) + 128;
    return YCC;
}
```

Figure 2: convertRGBtoYCC Version 1

Figure 2 shown above, shows what the conversion looked like during version 1. This iteration was very costly computationally because of the multiplications that had to occur for each pixel in the image. The equation for this was taken from the PowerPoint slide deck provided by the professor for the project guidelines.

```
uint8_t *convertRGBtoYCC(uint8_t red, uint8_t green, uint8_t blue) {
    uint8_t register *YCC = malloc(sizeof(int)*3);
    YCC[0] = 16 + (((red<<6)+(red<<1)+(green<<7)+green+(blue<<4)+(blue<<3)+blue)>>8);
    YCC[1] = 128 + ((-((red<<5)+(red<<2)+(red<<1))-((green<<6)+(green<<3)+(green<<1))+(blue<<7)-(blue<<4))>>8);
    YCC[2] = 128 + (((red<<7)-(red<<4)-((green<<6)+(green<<5)-(green<<1))-((blue<<4)+(blue<<1)))>>8);
    return YCC;
}
```

Figure 3: convertRGBtoYCC Version 2

Figure 3 shown above, shows the optimized version of convertRGBtoYCC. This version used an equation found at XXX which replaced all the multiplications with bit shifts and additions which saved CPU cycles compared to their corresponding multiplication [1].

The next optimization made was to change any other multiplication and divisions contained in the code to bit shifts. For example, when taking the average of the Cb and Cr values for 4 pixels, I changed the division by four into a bit shift right by two.

Optimizations Made in Assembly

No optimizations were made to the assembly code. I lacked the skills in this project to perform these optimizations, however this section outlines possible improvements that could be made with future work on this project.

ARM supports the use of saturating addition [2]. However, the C compiler which was used, GCC, is not smart enough to be able to specify saturating addition for ARM. This meant that for any addition that required saturating addition, I would have to go into the compiled assembly and change all `ADD` which required saturating add to `ADDS`. This change would no longer made the program portable to other architectures. If portability were to be a concern, this could've been achieved using an if statement in C where overflow is detected then the result is to set the to the maximum value for the type [3]. This would create a conditional branch, which I would attempt to avoid due to their cost.

After seeing the assembly produced by the compiled program, I noticed that there were a lot of loads occurring. A future improvement to the system that may be made, would be to change the design such that there were less loads occurring in the system. This suggestion is due to the fact that loads from memory are very costly to performance.

Hardware Supplement

No hardware supplement was created for this project. I felt that it was unnecessary given that reading the image I needed to be done in software. In the event that this wasn't the case and that the image was being provided to the converter as a stream of pixel information, the converter could be created using simple 8-bit adders and shifts. This is due to the fact that after optimizing the c code, the conversion and Cb and Cr downsampling was done purely using

additions and bit shifts. The color space conversion hardware diagram is available through an online resource which I used to assist with my learning of this project.

Performance and Cost Evaluation

This section will cover quantitative and qualitative performance testing performed on my project at each stage of development. Qualitative testing consists of running the program with a set of images and comparing the output image to the input. Quantitative testing uses measurable benchmarking to assess performance and cost of the system. Each set of tests will be discussed further in the sections following.

Testing Environment

All quantitative testing was performed the ARM virtual machine (VM) provided to the class by the professor. This VM was run using virtual manager on an Ubuntu operating system dual booted with Windows 10 on my personal desktop computer.

For ease of testing, all qualitative testing was performed on the Ubuntu OS itself. The reason for this was because the VM was command line based and lacked a GUI. Therefore, the decision was made to use the host OS to easily view the resulting images.

Qualitative Testing

I performed a handful of qualitative tests on my software which were used to judge the effectiveness of the color space conversion and image compression. The objective of these tests was to ensure that the conversion and compression didn't create any unintended or unexpected side effects. For these tests I used input images which I gathered online and compared them to the software's output image side by side.[1]

[1] It is important to note that this is not a perfect test to compare the qualitative accuracy of the software. This is because the color accuracy of the display being used to view the image comparisons may lack the range which differences in color may be detected by a person. More importantly, the ability to distinguish colors from one another varies from person to person, even as far as individual eyes. Therefore, one person may be able to distinguish color differences between two similar images while another may not.

Figure 4: Input and Output Comparison 2048 x 1357

The first test I performed was to compare and input image with the software's output image, shown in Figure 4: Input and Output Comparison 2048 x 1357. This test was performed on the functionally correct C version (Version 1) of the software, to ensure algorithmic correctness. As seen above, the color accuracy of the output image is satisfactory, however there are some off due to sharp lines and the compression technique used. Some color artifacts were found where there was a sharp line separating two drastically different colors. Reference Figure 5: Image Color Artifact Example 204 x 135. The cause of this was found to be due to the compression of the Cb and Cr values in a 2 by 2 space by averaging the four values of each. No exist cause could be narrowed down to be fixed, however I suspect using a more advanced down sampling algorithm may fix the issue. It can also be seen in both Figure 4 and Figure 5 that the color accuracy between the original and output images are not exact. At a glance the images look the same, yet upon further inspection of the images some variation in hue and saturation can be seen. Figure 5 has the images mirrored to highlight the slight color difference in the wooden boards of the building.

Figure 5: Image Color Artifact Example 204 x 135

Overall, I am satisfied with the color accuracy of the resulting image and hope that in the future I can come back to this project and improve the down sampling of the Cr and Cb values.

Quantitative Testing

C Profiling Using GPROF

Profiling on the two versions of this software was performed using gprof. This profiling was done twice, once on the Ubuntu host machine to gauge approximate improvements and then done on the ARM VM. The reason for profiling on the host was to show how the differences in architecture changes the performance increases made by optimizing the C code.

The table below shows the difference in version 1 and version 2 run on the Ubuntu host. There can be seen a noticeable change in the convertRGBtoYCC method, from 10.80 ns/call down to 7.20 ns/call.

Version 1						
%	Cumulative	Self (s)		Self	Total	
Time	Seconds	Seconds	Calls	ns/call	ns/call	Name
54.60	0.06	0.06				Main
27.30	0.09	0.03	2781184	10.80	10.80	convertRGBtoYCC
9.10	0.10	0.01	2781184	3.60	3.60	convertYCCtoRGB
9.10	0.11	0.01	1390592	7.20	7.20	avg
Version 2						
%	Cumulative	Self		Self	Total	
Time	Seconds	Seconds	Calls	ns/call	ns/call	Name
37.54	0.03	0.03				Main
25.02	0.05	0.02	2781184	7.20	7.20	convertRGBtoYCC
25.02	0.07	0.02	2781184	3.60	3.60	convertYCCtoRGB
9.10	0.11	0.01	1390592	7.20	7.20	avg

The next table shows the differences in performance on the ARM VM. It shows a drastic increase in performance after the optimizations made.

Version 1						
%	Cumulative	Self		Self	Total	
Time	Seconds	Seconds	Calls	ns/call	ns/call	Name
60.31	1.91	1.91				Main
33.15	2.96	1.05	2781194	377.89	377.89	convertRGBtoYCC
5.37	3.13	0.17	2781194	67.18	61.18	convertYCCtoRGB
1.26	3.17	0.04	1390582	28.79	28.79	Avg
Version 2						
%	Cumulative	Self		Self	Total	
Time	Seconds	Seconds	Calls	ns/call	ns/call	Name
71.01	1.44	1.44				Main
17.26	1.79	0.25	2791184	125.97	125.97	convertRGBtoYCC
10.36	2.00	0.21	2791184	61.18	61.18	convertYCCtoRGB
1.48	2.03	0.03	1390582	21.60	21.60	Avg

These results show that the optimizations I made which were mentioned in sections above, had an impact on the performance of the system. The time per call of convertRGBtoYCC was reduced over 60%. This was not an insignificant improvement, and therefore I believe that it was well worth the development time to implement.

Image File Size Comparisons

Theoretically, by compressing the Cb and Cr values for each 2 by 2 pixels, the overall size of the image could be reduced for transmission and storage of the YCbCr representation of the image. This is because only 1 set would be necessary to send or store for each set of 4 pixels, rather than a set for each pixel. This would result in an approximate 58% decrease in size dedicated for pixel information for 4 pixels. Due to the need to replicate these compressed values in the PPM format, these decreased image sizes could not be observed in the output images.

Conclusions

Over the duration of this course and project, I learned a lot regarding developing software for the use in an embedded system. The concepts learned includes what embedded systems are, what constraints exist in an embedded software environment along with techniques that can be used to work around them, and hands on experience with developing software for embedded systems. I was able to achieve the goal of the project, even with the challenges I faced. This project shows that the concepts taught in the course work for improving the

performance of embedded systems, where the optimizations taught in class and used within this project created a significant and noticeable improvement on performance. On the other hand, I still have lots to learn, practice and perfect in this topic space. With this course and project ending, I hope to learn further and, in the future, come back to this project and improve upon it so that I can continue to develop my skills as a software developer and have something tangible to show improvements in my capabilities.

Bibliography

[1] N. Campos, "RGB to YCbCr conversion," 21 August 2019. [Online]. Available: https://sistenix.com/rgb2ycbcr.html. [Accessed 1 July 2019].

[2] "ARMv7-M Architecture Reference Manual," February 2010. [Online]. Available: https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf. [Accessed 1 July 2019].

[3] MSalters, "Stack Overflow Unsigned Saturating Addition in C," 3 October 2008. [Online]. Available: https://stackoverflow.com/a/166393/10592529. [Accessed July 2019].