ECE 785
TOPICS IN ADVANCED COMPUTER DESIGN
PROJECT 3 REPORT

Name: Cristian Hellmer

Unity ID: crhellme

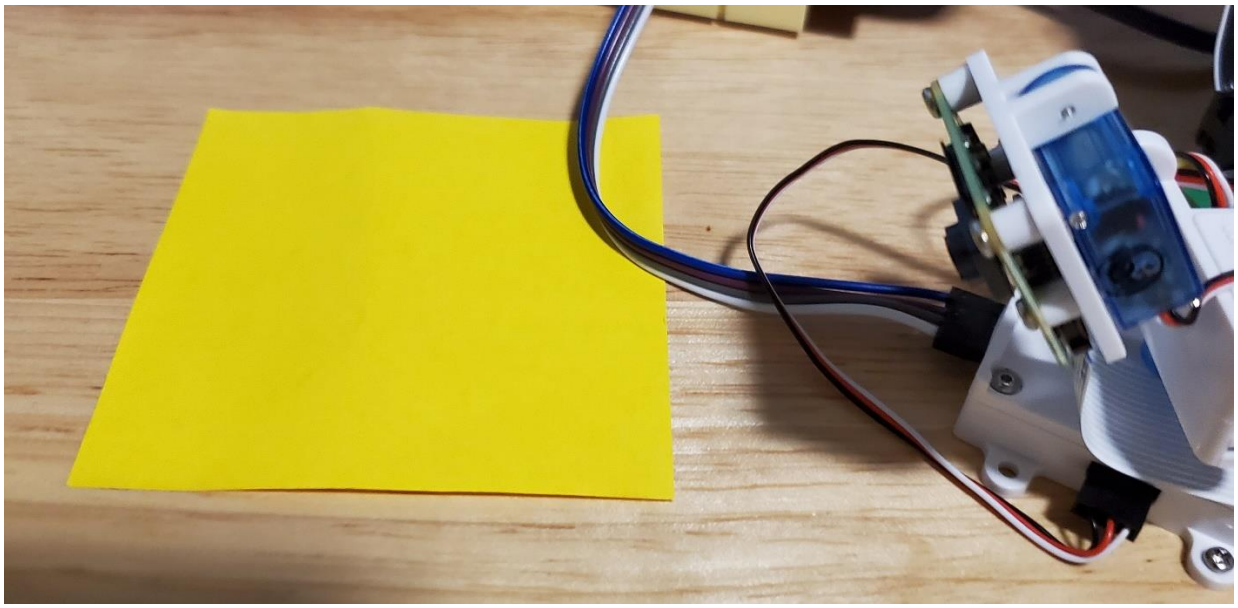
Introduction

In this project, we were tasked with optimizing a program which uses a gimbal-mounted Raspberry Pi Camera to track a designated object and stabilize the image being received by the camera so that the object remains in the center of the screen. The program has the ability to stabilize the image both digitally and using servos, but this project focused on optimizing the digital stabilization and image tracking capabilities of the software. Specifically, we were optimizing the speed of the color matching function and the functions which it calls.

The project was divided into 2 parts, Part A and Part B. Part A optimizations were to focus on scalar optimizations and Part B was to focus on applying vectorization. The vectorization leverages the Neon Advanced SIMD capabilities found on the Cortex A72 processor. Using 128-bit wide vector registers, the program was able to process 8 pixels in parallel as opposed to the previous implementation of 1 pixel at a time. Versions 1 though 9 of the code are from part A and version 10 is the result of part B.

To keep testing consistent, the camera, the color to match, and the object which it was tracking were all kept stationary and consistent for all of the test runs while recording performance. This was done because the average frame processing time varies heavily based on how much of the image matches the color being used to track the object.

Camera Setup



Average Frame Time Optimization

Code Version	Average Frame Time Without Stabilization (ms)	Average Frame Time With Stabilization (ms)	Speed-Up Factor Without Stabilization	Speed-Up Factor With Stabilization	Time Spent on Optimization (hours:minutes)	Cumulative Time Spent on Optimizations (hours:minutes)
Initial Code	8.935	11.250			0:00	0:00
1	8.690	11.292	1.028	0.996	0:20	0:20
2	7.846	10.095	1.108	1.119	3:00	3:20
3	7.621	10.148	1.030	0.995	0:30	3:50
4	6.944	9.521	1.097	1.066	0:30	4:20
5	6.186	8.631	1.123	1.103	1:00	5:20
6	5.987	8.765	1.033	0.985	0:20	5:40
7	4.996	8.092	1.199	1.083	0:30	6:10
8	3.190	5.897	1.566	1.372	0:30	6:40
9	2.648	5.167	1.205	1.141	0:40	7:20
10	2.209	4.770	1.199	1.083	5:00	12:20

Part A Optimizations

Code version 1 optimization: The optimization performed for this version of the code was editing the CMakeLists.txt file with compiler flags which would guide the compiler to generate more optimized object code. The flags which were added were "-Ofast", "-funroll-loops", "-mcpu=cortex-a72", "-mfpu=crypto-neon-fp-armv8", "-mfloat-abi=hard", and "-funsafe-math-optimizations."

Code version 2 optimization: The optimization performed for this version of the code was eliminating the memcpy() call in the video_buffer_callback function. Instead of copying the buffer taken in to another buffer to be worked on, the buffer is directly worked on and sent to the preview input port. I chose to optimize this because before this optimization memcpy() was taking a considerable amount of time the program execution time. After optimization, memcpy () takes a negligible amount of time.

Code version 3 optimization: The optimization performed for this version of the code was replacing the w and half_w variables being used in the Set_Pixel, Set_Pixel_yuv, and Get_Pixel_yuv functions with their known numerical values. I chose this optimization because I thought that loading constants into registers would be faster than looking up a variable's value in memory.

Code version 4 optimization: The optimization performed for this version of the code was removing unnecessary comparison code in the Draw_Line function. I chose this optimization because I noticed there was a lot of code in the Draw_Line function that was not applicable to the functions which call Draw_Line. Also, Draw_Line takes a considerable amount of time in the program.

Code version 5 optimization: The optimization performed for this version of the code was moving the code in the Get_Pixel_yuv function inside the find_chroma_matches function instead of calling Get_Pixel_yuv. I did this because there was a high percentage of execution time in Get_Pixel_yuv spent loading the calculated valued into memory to be sent back to the find_chroma_matches. I eliminated this by not needed to pass variables when calling the function.

Code version 6 optimization: The optimization performed for this version of the code was replacing the "i->w" and "i->h" with their known numerical values used in the for loops in find_chroma_matches. I chose this optimization because it seemed wasteful to me to look up these values in memory when they do not change during the duration of the program.

Code version 7 optimization: The optimization performed for this version of the code was moving the code found in the Sq_UV_Difference_yuv function to the find_chroma_matches instead of calling the function Sq_UV_Difference_yuv. I did this because the Sq_UV_Difference_yuv function reads values from memory which were passed into the function. These memory lookups could be eliminated by moving the code to the find_chroma_matches function so that variables don't need to be passed.

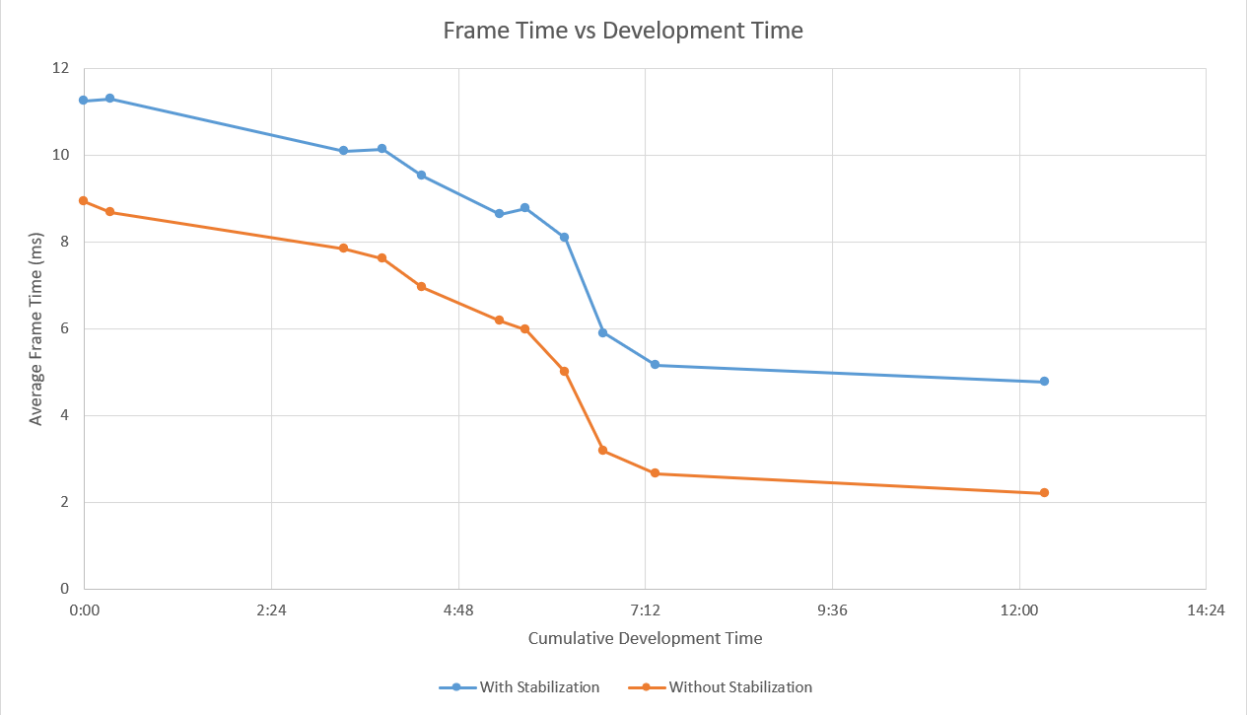
Code version 8 optimization: The optimization performed for this version of the code was changing the matching pixel shading code in the find_chroma_matches function from having 2 Draw_Line function calls to using the function Set_Pixel_yuv to change the pixel currently being analyzed to pink. I chose to change this because I noticed that the Draw_Line calls were changing pixels needlessly when they were not required to shade in the matching colors with a separation of 2.

Code version 9 optimization: The optimization performed for this version of the code was replacing the Set_Pixel_yuv call in find_chroma_matches with the actual code found in Set_Pixel_yuv. I chose this optimization in order to eliminate looking up the color to set the pixel from memory when we always set it to pink when the function is called inside find_chroma_matches.

Part B Optimizations

Code version 10 optimization: The optimization performed for this version of the code was applying vectorization to the implementation of find_chroma_matches using Neon Advanced SIMD intrinsics. I chose this optimization because the operations applied to the pixels in the image passed to the find_chroma_matches function had the opportunity to be processed in parallel.

Analysis and Discussion of Results



The optimization which improved performance the most was changing the matching pixel shading code in the find_chroma_matches function from having 2 Draw_Line function calls to using the function Set_Pixel_yuv to change the pixel being analyzed.

Retrospective

During this project, it was interesting to see what aspects of the code took the most amount of time and how to focus on these aspects to get the most improvement in performance. Before this project, I hadn't had the opportunity to use a tool like perf to see which object code lines take a large percentage of a function's runtime. Oftentimes, the memory accesses in the program would take a disproportionate amount of time compared to other instructions. Because of this, much of the time spent optimizing was trying to find how to eliminate these memory accesses as much as possible. In the future when I need to focus on improving the speed of a project, I'll definitely keep this in mind. Part B was a good introduction to applying vectorization and parallelism to my code. It was intriguing to see the changes to code which are necessary to enable effective vectorization of the code, such as eliminating the conditional statement in the find_chroma_matches function.