Lab 2 Report

Adrian Saldana & Noor Fakih

*Board’s host name:* ee180-23z.stanford.edu

**Assignment/Problem Description:**

Lab 2 required us to optimize a C++ implementation of Sobel filter for an 800MHz ARM Cotex-A9 processor on the Zedboard development board. The goal of the assignment was to experience how having an understanding of computer architecture can result in highly efficient code and the different methods to do so such as vectorizing and strength reduction. In the first part of the lab, we optimize the single-threaded performance of the Sobel filter. In the second part, we build upon our single-thread optimization to create a multithreaded version that uses thread-level parallelism.

**Discussion**:

*Solution/Program Description*

The program takes the input video and applies a Sobel filter to each frame of the video. Each frame gets passed to grayScale(), which returns an approximation of the gradient of the image density function and is commonly used to perform edge detection on images. The grayscale frame is then passed to sobelCalc which then performs a 2-D convolution, transforming each pixel into a weighted average of the 3x3 grid of pixels around it.

The multi thread version has two threads working in parallel. Each thread will process half of each frame, and come together to produce a full frame.

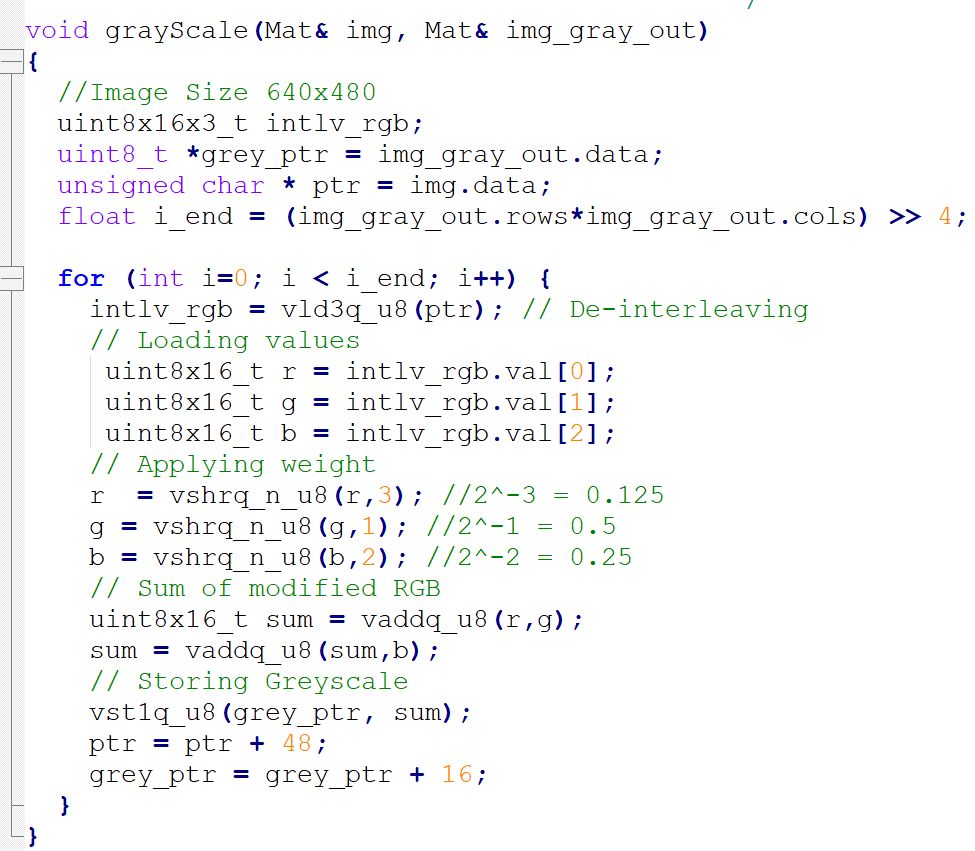
*Known Bugs and/or Errors*

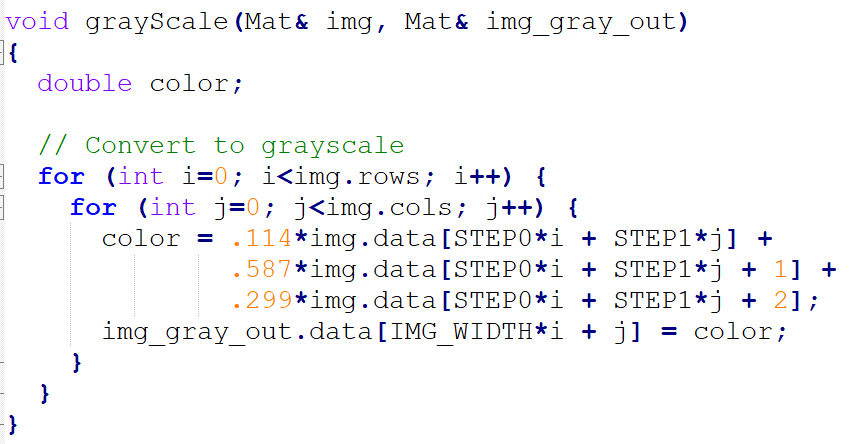
Since we were given a skeleton starter code for the lab, there was already a procedure for ensuring proper image capture and the mechanism for viewing the filtered output. The starter code already had an implementation for a single thread sobel filter, it just was not optimized. Our implementation of the assignment does not have bugs.

**Optimization and Results**

Starting fps with no optimization - 3

*Part 1: Single Thread Case*

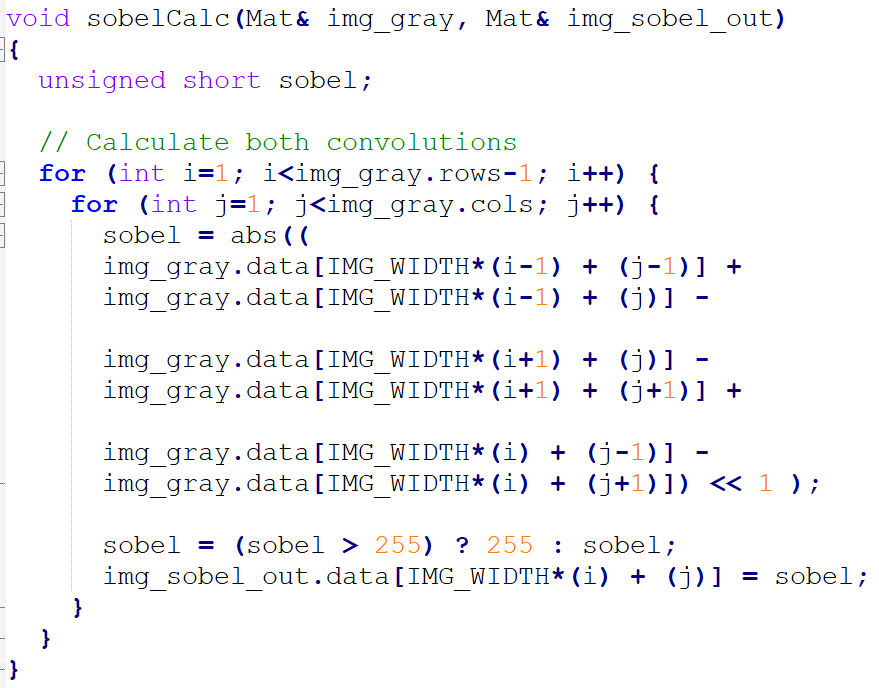
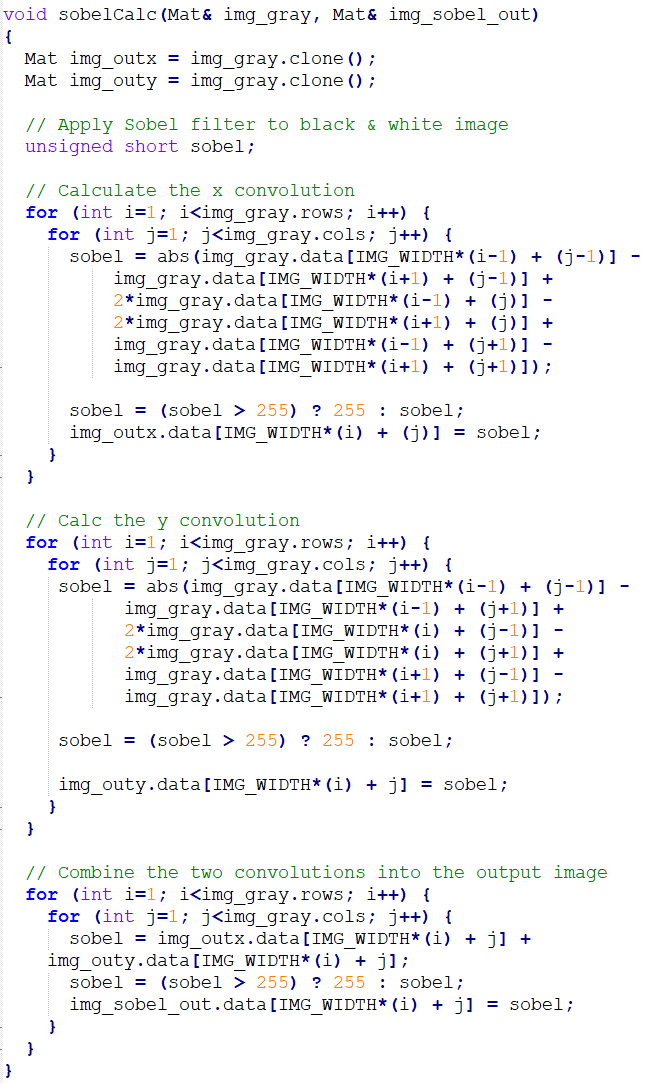
Original Optimized

Inefficient operations:

1. multiplying with a float
2. Using floats
3. Repeatedly multiplying and recalculating the same value
4. Double for-loop and only processing one pixel at a time

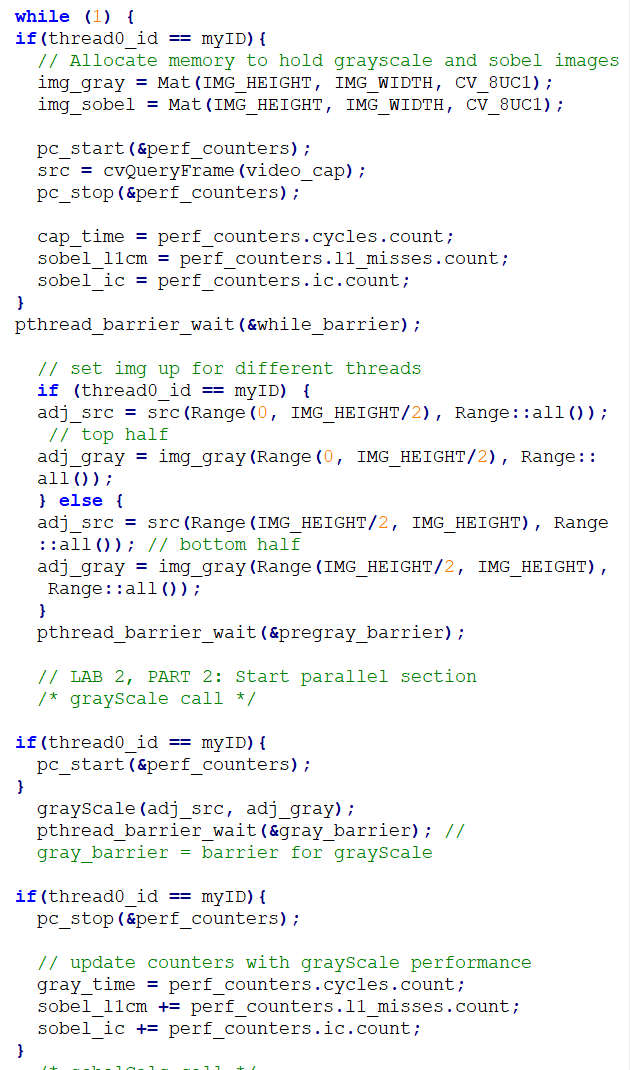
We restructured the double for-loop into a single for-loop for easier pointer manipulation. We then used neon vectors and intrinsics so we can process 16 pixels at a time. The neon intrinsic vld3q de-interleaves the 8 bit RGB values in the source img into three vectors, each with 16 elements. We apply the weight by bit-shifting so we can minimize the number of bits needed to represent the rgb values and maximize the number of pixels we can process at a time. A bit shift to the right is equivalent to multiplying by ½ which is approximately 0.587 and is significantly faster operation than float multiplication.

Original Optimized

Inefficient operations:

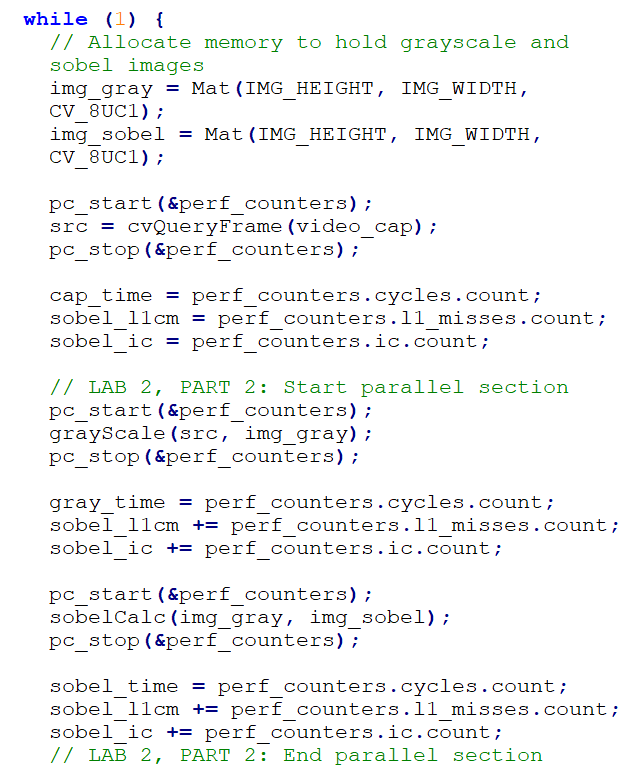
1. 3 double for-loops when you only need one
2. Repetitive code
3. Separately calculating x and y convolution
4. Making two Mats
5. Repeatedly accessing + writing to img\_gray

Instead of calculating the x and y convolution separately, writing the data in separate structs, then adding them together and writing that to output, we did that all at once. In our single double for-loop, we calculate the sum of the x and y convolution of the image, check once that no pixels are over 255 (max value stored in 8 bits), and write it to output. There is significantly less multiplication, repetitive code and loading/writing data in our version as well.



*Part 2: Multi Thread Case*

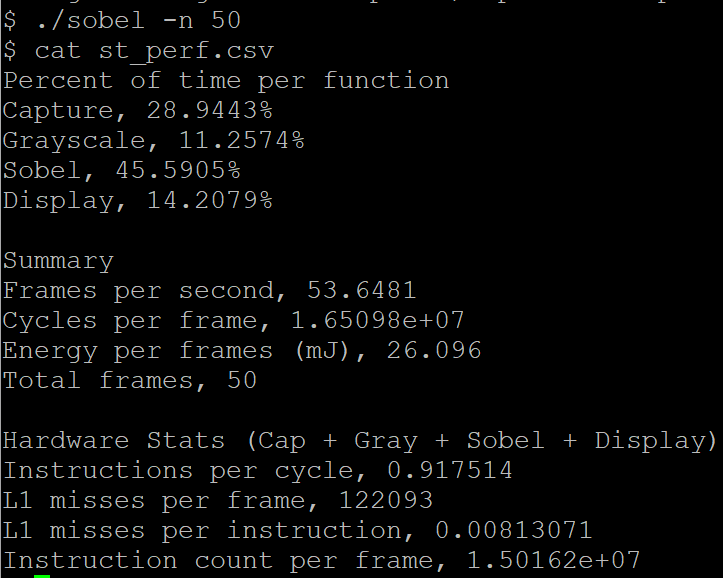
Original Optimized ->



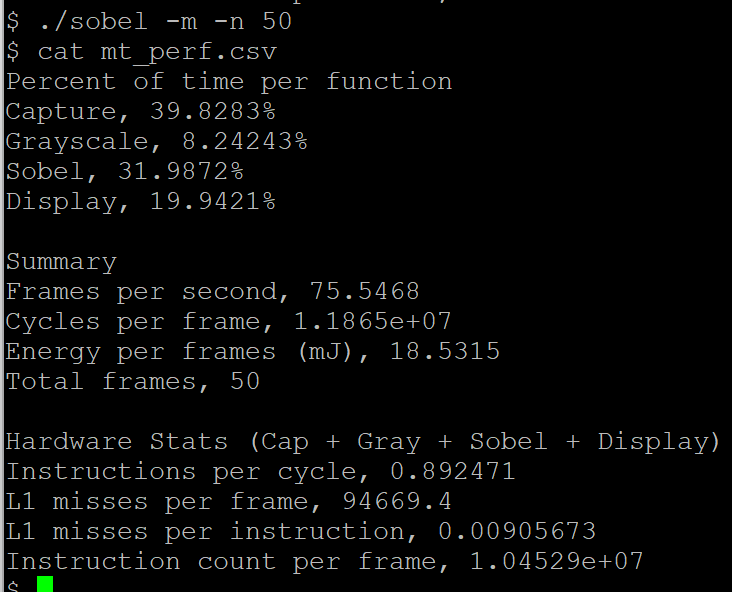
We were provided with a version of sobel\_mt.cpp that launched two threads, killed one of them instantly and processed the image with just one thread. In our implementation, we use both threads to process each frame of the video - thread0 processes the top half of the image and thread1 processes the bottom half. We also ensure that only one thread (thread0) does things like interact with the performance counters, captures the source image, and writes out to the excel sheet. We also added barriers periodically to ensure the threads move in parallel.

*Results:*

Single thread optimized results:



Multi thread optimized results:



**Lessons Learned/Epilogue:**

1. General methods for optimization
   1. Loop unrolling
   2. Strength reduction
   3. Condense repetitive code
2. Multithreading
3. Vectorization