Concurrent Photo Filters in Go
Kailey Cozart

Overview

For my project, I created black and white, sepia, and negative filters in Go. Each filter was a function that was run concurrently with the other goroutines. The code, in its current state, will work if the image you are filtering is named Original.jpg and if it is in the same folder as final.go.

Other Possible Uses of Concurrency

This project uses concurrency to do different tasks. Some other uses of concurrency would be using CUDA to do an embarrassingly parallel kernel where each thread maps to a single pixel and completes the computation for that individual pixel.

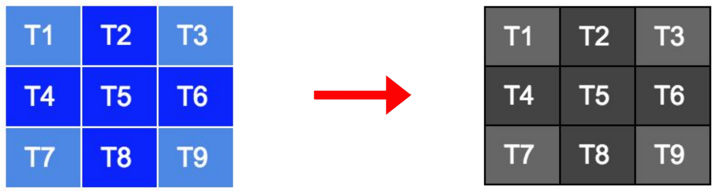


Image Format

If you would like to apply this filter to png or gif images, simply uncomment the “image/png” or “image/gif” packet and comment out the “image/jpg” packet. Also note that all jpeg.Decode() and jpeg.Encode() statements must be changed to the appropriate image format.

Possible Errors

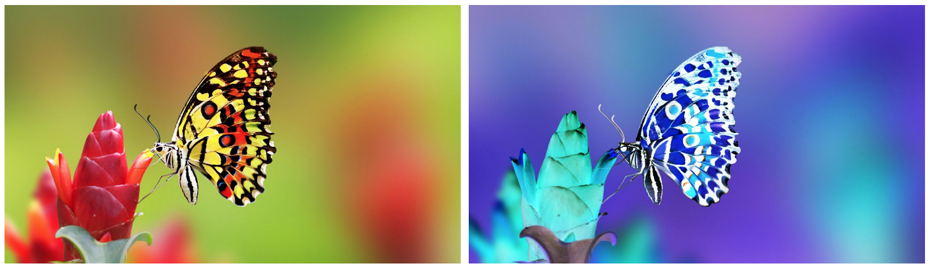
If there is no image named Original.jpg in the same folder as the final.go file, the program will terminate with an error code. Note that Original.jpeg is not equivalent to Original.jpg. Original.jpg will work with the code as-is, but Original.jpeg will not.

Black and White Filter

The most basic black and white filter takes RGB values and averages them, replacing all values with that average. This, however, can result in an unsatisfactory filter. Therefore, the weights of 0.229, 0.587, and 0.114 are added to the RGB values, respectively, in order to create a more high-quality filter. The color.Grey library function is used for the black and white photo filter. Below is an example created by this filter:

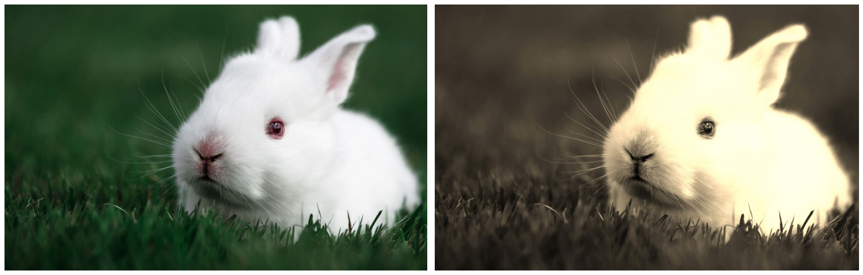
Negative Filter

The negative filter is the most simple computationally. Each existing RGB value is subtracted from the maximum possible value. Typically, the maximum value is thought to be 255, but, for 16-bit values, 65535 is the maximum. First, the img.At function is used with the loop values of x and y to find the correct pixel. Then, the img.RGBA function is used to get the original r, g, and b values. Finally, after the subtraction is executed, the new RGB values must be added to the new image object with color.RGBA. These RGB values must be unsigned ints that are 8 bits long. Because we are truncating the existing 16 bit values to be 8 bits long, we must shift the bits 8 positions to the right because the top 8 bits are the ones with the important color values. If we don't shift them, then the filtered image will end up being filled with random RGB values. Below is an example created by this filter:



Sepia Filter

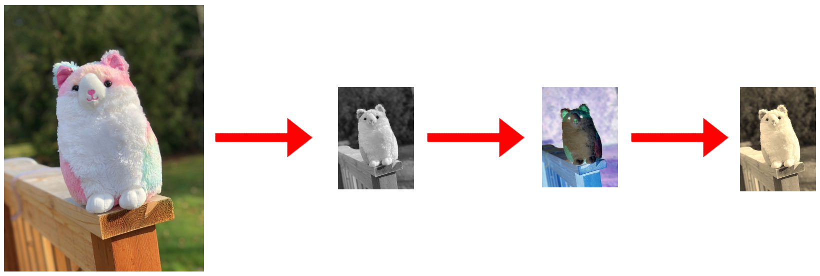
The sepia filter uses 9 constant floating point values that are multiplied by each RGB value. These are predefined constants that have been recommended by Microsoft for the sepia filter. After the RGB values are multiplied by the floats, the computed value must be rounded to the nearest integer using the math.Round function. The int function must then be used to cast the float value into an int. A final important step is to check that each new RGB value is not above 65535. If the value is above 65535, it will overflow and cause the image to have various incorrect pixels. And values above 65535 must simply be replaced with 65535. Below is an example created by this filter:



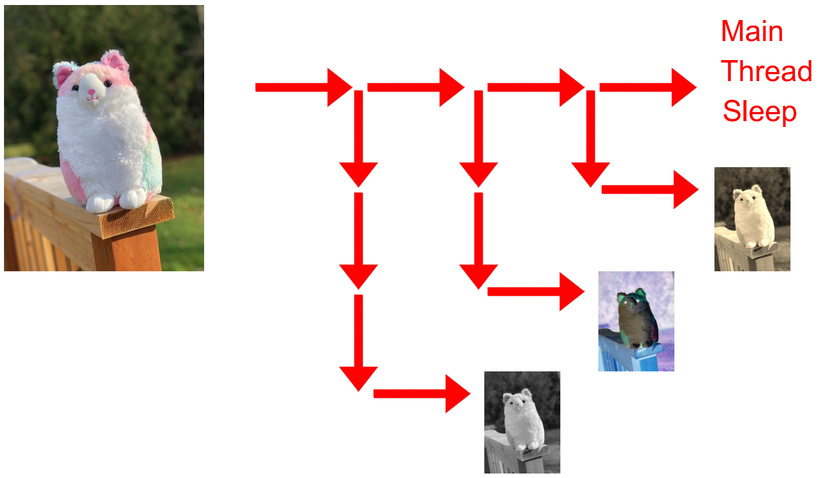
Benefits of Concurrency

Running these three filters concurrently speeds up the code, especially when high-quality images are being filtered. One note is that the code includes a time.Sleep function at the end of main. This is because main spins off the three concurrent threads, reaches the end of main, and then terminates. Go, unlike C++ or CUDA, does not rejoin the concurrent threads. Therefore, adding a wait period at the end of the main function allows the concurrent threads to finish their tasks before main terminates.

Serial Code:



Concurrent Code:


Main
Thread
Sleep