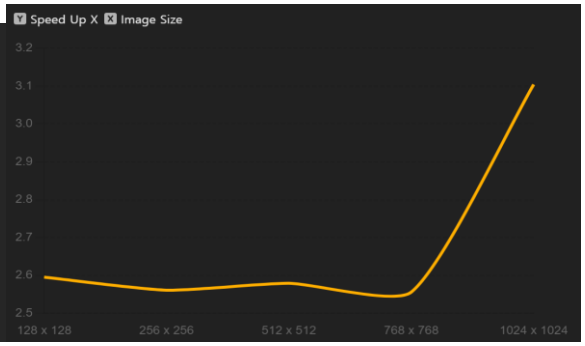
HW2 REPORT

-한웅재(202011226)

1] IMPLEMENTATION RESULTS

Image	Speed Up
128*128	2.595334
256*256	2.561011
512*512	2.579204
768*768	2.555854
1024*1024	3.229826



The table and graph above is the data of speed up according to image size. For images size of 768, 512, 256 and 128, speed up shows about 2.6 times of baseline filter. However, the image with a size of 1024 x 1024 showed a speed up of 3.1 times. These results demonstrate that a considerable image size is required for code optimization to effectively speed up the process.

2] OPTIMIZATION APPROACH

1. Function call overhead for convolution function

   The first thing I considered in optimizing the code was the issue with the Convolution function. If the Convolution function and the filter function are implemented separately, calling Convolution from the filter function can lead to overhead, resulting in decreased performance. Therefore, our initial thought was to directly integrate the Convolution function into the filter function. The speed up achieved through this change was about 1.5 times.

2. Loop Unrolling with convolution

   After optimizing the code considering function call overhead, we identified an issue: there were too many for loops in the optimized filter function. Therefore, we started by inspecting the innermost loops first, and decided to apply loop unrolling.

   Originally, the code for performing convolution consisted of two nested for loops, moving along the x and y axes of the input and shifting the filter along its x and y axes to calculate the convolution for each pixel's r, g, b values. With a 3x3 filter, this results in nine calculations. We unrolled these loops completely, eliminating the loops for all nine elements of the filter and re-writing the code accordingly. The speed up achieved from this modification was about 2.7 times.

3. Code motion

   After implementing loop unrolling, we observed several variables within the loop that did not change. For example, each element of the filter from 0 to 9 is used in every loop but does not change its value. Therefore, we moved these to outside of the loop. Similarly, values like xm1 in the y loop also did not change, so we moved them outside of the loop as well. With these changes, I achieve a maximum speed up of 3.2.

4. Chache friendly

In the filter function, when accessing the input matrix to perform convolution, we changed the order of the base filter's loops by placing the y loop inside the x loop. This implementation was chosen to make the access more cache-friendly. We aimed to increase the hit rate by accessing the same row, which helped to enhance cache utilization. The speed up achieved through this adjustment was around 0.5.

5. Others

I removed all the code related to memory management such as memset and malloc first. In the original code, allocating memory with malloc for each loop iteration and then freeing it, or initializing the fields of the Pixel structure to 0 was unnecessary. Therefore, by removing these, I eliminated needless memory allocation and initialization. This change likely resulted in a speed up of about 0.3 times.

Additionally, the filter was initially defined as a float *, containing float values. However, since operations with integers are typically faster than those with floating-point numbers, I converted all the types in the filter to integers and used these integer values instead.

6. Failed Try

Firstly, I attempted to replace inequality operations with bitwise operations, but this did not result in any noticeable speed difference. Secondly, I tried copying a specific block size from the input matrix in advance and then accessing this block matrix to perform operations. However, this approach did not show good performance due to the significant overhead from memcpy.