2024 system programming
Assignment 2. Efficient Program Implementation

Student ID: 202011046
name: 김소혜

I. Results

Img_1024.bmp	2.288007	2.209422	2.301259	2.252887	2.245279
Speedup	2.234077	2.250467	2.253279	2.220384	2.269750
Img_768.bmp	1.887893	1.869957	1.902550	1.877021	1.899201
Speedup	1.895034	1.894235	1.888379	1.879447	1.884249
Img_512.bmp	1.919301	1,911447	1.902333	1.879286	1.871255
Speedup	1.904846	1.871654	1.883611	1.880853	1.872893

II. Optimization approach

1. I moved the ‘malloc’ and ‘free’ functions outside the ‘for loop’. Although I wanted to change the values from the Pixel type and remove the memory allocation code, I did not modify this part to maintain the basic structure of the function (e.g., return values)
(Img_1024.bmp) Speedup : 1.0x -> 1.1x

2. I replaced functions like sizeof(p) with numeric values to reduce overhead.
(Img_1024.bmp) Speedup : 1.1x -> 1.1x~1.2x

3. For calculations that are repeatedly used within the for loop, I precomputed them and assigned the results to variables for reuse. Additionally, I replaced multiplication operations with addition and bitwise operations wherever possible.
(Img_1024.bmp) Speedup : 1.1x~1.2x -> 1.2x

4. Since this is C code and in many cases memory is organized in row-major order, I adjusted the order of the for loops so that the inner loop iterates over x (columns).
(Img_1024.bmp) Speedup : 1.2x -> 1.4x~1.5x

5. I used loop unrolling. Specifically, in ‘filter_optimized’ part, since the width is a multiple of 32, I performed loop unrolling for 4, 8, and 16 iterations. The execution results alternated between speedups of about 2.0xx and 0.8xx. After further repeated testing, I found that unrolling the loop 16 times provided a more stable speedup of 2.0x, so I decided to use 16 iterations for the loop unrolling. Additionally, in ‘convolution’ part, loop unrolling allowed the removal of unnecessary conditional statements and calculations. During this process, I also assigned repeatedly used values to variables for reuse.
(Img_1024.bmp) Speedup : 1.4x~1.5x -> 2.0x

6. When comparing the computation speeds of data types such as int, float, and double, it is obvious that int was faster than the other data types. Therefore, I used type casting to convert float and double types to int. Even after type casting, the execution results were normal. Considering that the computational resources for int and float might differ, I

tried to alternate their placement to allow simultaneous computation. However, this did not show significant effects (The speedup values were unstable. Due to fluctuations in the values, it is difficult to say precisely, but they varied roughly between 2.0x and 2.1x), and in my view, the speedup was slightly higher when all were converted to int. Thus, I changed all to int and also removed some unnecessary type castings.
(Img_1024.bmp) Speedup : 2.0x -> 2.2x