System Programming Hw2

Student Name: 우희담

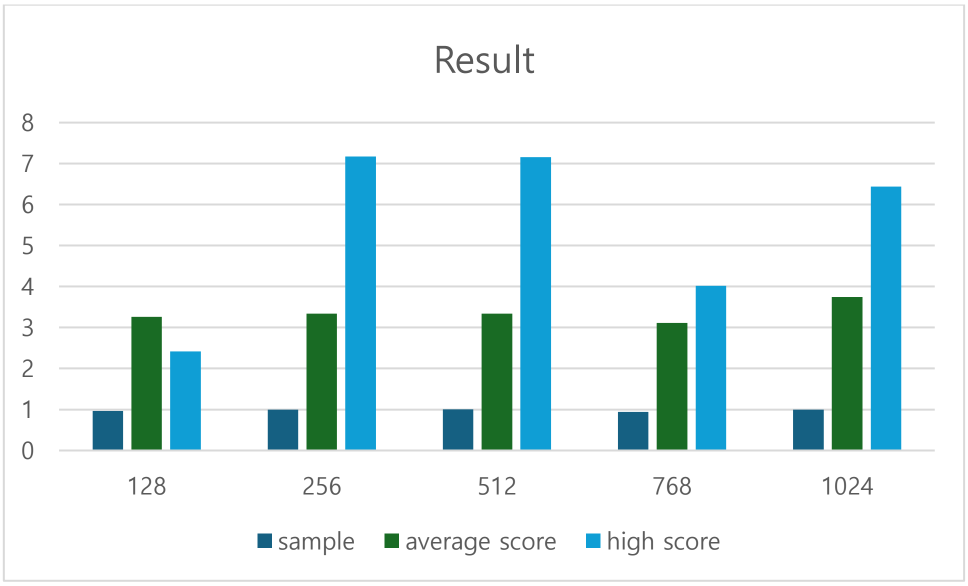
Student ID: 202211121

Section 1: Result

As a result of completing the assignment, I was able to achieve the results shown in the table below. The graph visually represents the data from the table, with results from left to right for image sizes 128, 256, 512, 768, and 1024 respectively. There was variation in each attempt. So I calculated the average of 5 attempts and attached a photo file of the highest score result.(result.xlsx and .png files)

	128	256	512	768	1024
sample code	0.967552	0.994981	1.005099	0.943553	0.994235
average score	3.264039	3.3364854	3.3358418	3.1096192	3.741693
high score	**2.417124**	**7.174989**	**7.157078**	**4.019243**	**6.437964**

*The average score is the average of 5 attempts



Section 2: Optimization Approach

(Note: I could not record the individual speedup results for each code modification, so I could not determine the specific impact of each approach.)

1. **Elimination of Dynamic Memory Allocation**

- Upon reviewing the original code, I observed frequent calls to malloc and free for each pixel processing, which introduced significant overhead and degraded performance. Therefore, I removed these dynamic memory allocations to improve efficiency.

2. **Distinction between filter_boundary and filter_center**
    - The original filter_optimized function applied convolution to all pixels, including those on image boundaries with zero padding. This unnecessary computation was prevalent. To address this, I distinguished functions to handle zero padding (filter_boundary) from those handling non-zero padded areas (filter_center). This separation eliminates the need for boundary condition checks in filter_center, thereby reducing condition checks within the loop and speeding up pixel processing. The filter_boundary function is further split into handling top-bottom boundaries and left-right boundaries.

3. **Loop Unrolling**
    - From supplementary lectures, I learned that loop unrolling is an optimization technique that reduces the number of iterations in a loop, thus reducing the overhead of loop control and improving the efficiency of the program. Additional research suggested that this technique not only minimizes iteration count but also allows compilers more opportunities for code optimization. Therefore, I applied loop unrolling as follows. The original convolution function in the sample code applied a 3x3 filter using nested loops for each pixel and its surroundings. To eliminate this nested loop, I explicitly coded each scenario as shown below:

```
// -1, -1
            p = input[(x - 1) + (y - 1) * width];
            f = filter[0];
            r += p.r * f;
            g += p.g * f;
            b += p.b * f;
```

Unsuccessful Strategies

- **Threading without Makefile Usage**: I researched relevant methods but could not find an appropriate implementation.
- **Multiplication to Shift Operation in Convolution**: Attempting to replace multiplication with shift operations in the convolution function altered the results and did not produce the expected speedup.
- **Using goto for Loop Implementation**: I tried implementing loops with goto, which slightly increased the speedup by 0.02 for sizes 128 and 256(2.395771), 512(2.390440). However, for sizes 768(2.388625), and 1024(2.390440), it actually decreased performance by 0.02, leading me to reject this approach.