시스템 프로그래밍 HW2 Report

202111062 박동진

1. Optimization approach

In order to optimize the performance of the program that applies a 3x3 convolutional filter to the BMP image, the optimization attention points were selected as follows and the optimization code was written accordingly.

(1) Remove unnecessary dynamic memory allocation

In the original code, the 'Pixel' structure was dynamically allocated and released for each iteration through 'convolution'. This was expected to degrade performance due to overheads for memory allocation and release. Therefore, this iterative dynamic memory allocation was removed and implemented so that the 'convolution' function directly returns the 'Pixel' object. This will eliminate the repetitive overhead due to memory allocation and release and will lead to a better optimization code than before.

(2) Optimizing Border Processing

In the original code, boundary inspection was performed on all pixels. It was thought that this would lead to performance degradation because unnecessary condition inspection is performed even for pixels that do not belong to the boundary. Therefore, as a method for optimization, a method of separating and processing the boundary and the internal area was applied. In the case of the internal area, the boundary inspection was omitted by additionally implementing the 'convolution_no_boundaries' function so that a separate boundary inspection was not performed. As a result, it is expected that the condition inspection overhead of the internal area will be reduced and it can act as an optimization factor.

```
// 경계 처리 없는 convolution 함수
static Pixel convolution_no_boundaries(
        Pixel* input, int x, int y, int width, float* filter) {
    double r = 0;
    double g = 0;
    double b = 0;

    for (int dy = -1; dy <= 1; ++dy) {
        int yy = y + dy;
        for (int dx = -1; dx <= 1; ++dx) {
            int xx = x + dx;

            int idx = xx + yy * width;
            int fidx = (dx + 1) + (dy + 1) * 3;

            r += input[idx].r * filter[fidx];
            g += input[idx].g * filter[fidx];
            b += input[idx].b * filter[fidx];
        }
    }
```

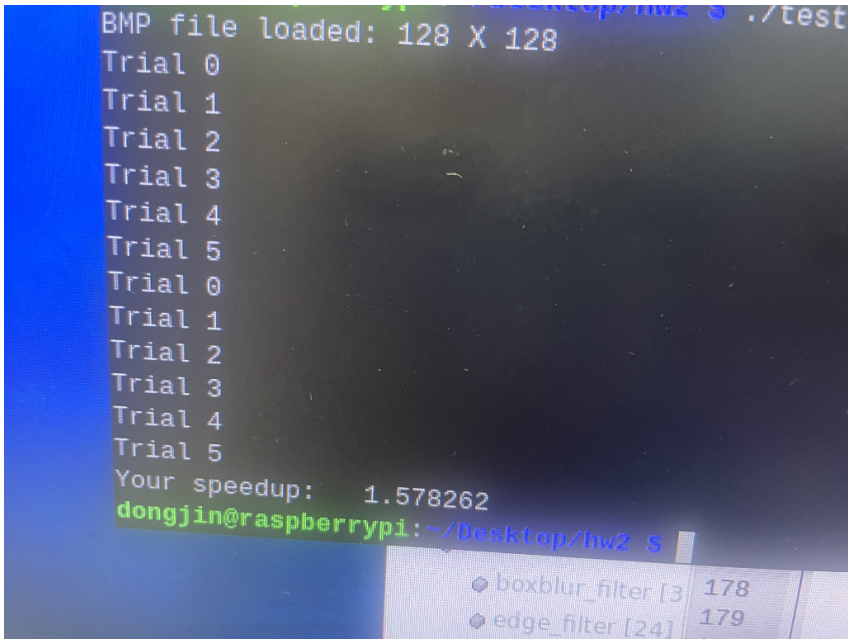
```
// 내부 영역 (경계가 아닌 부분) 처리
for (int y = 1; y < height - 1; ++y) {
    for (int x = 1; x < width - 1; ++x) {
        output[x + y * width] = convolution_no_boundaries(input, x, y, width, filter);
    }
}

// 상단 경계 처리
for (int x = 0; x < width; ++x) {
    output[x] = convolution(input, x, 0, width, height, filter);
}

// 하단 경계 처리
for (int x = 0; x < width; ++x) {
    output[x + (height - 1) * width] = convolution(input, x, height - 1, width, height, filter);
}

// 좌측 경계 처리
for (int y = 0; y < height; ++y) {
    output[y * width] = convolution(input, 0, y, width, height, filter);
}

// 우측 경계 처리
for (int y = 0; y < height; ++y) {
    output[(width - 1) + y * width] = convolution(input, width - 1, y, width, height, filter);
}
```

(3) loop optimization

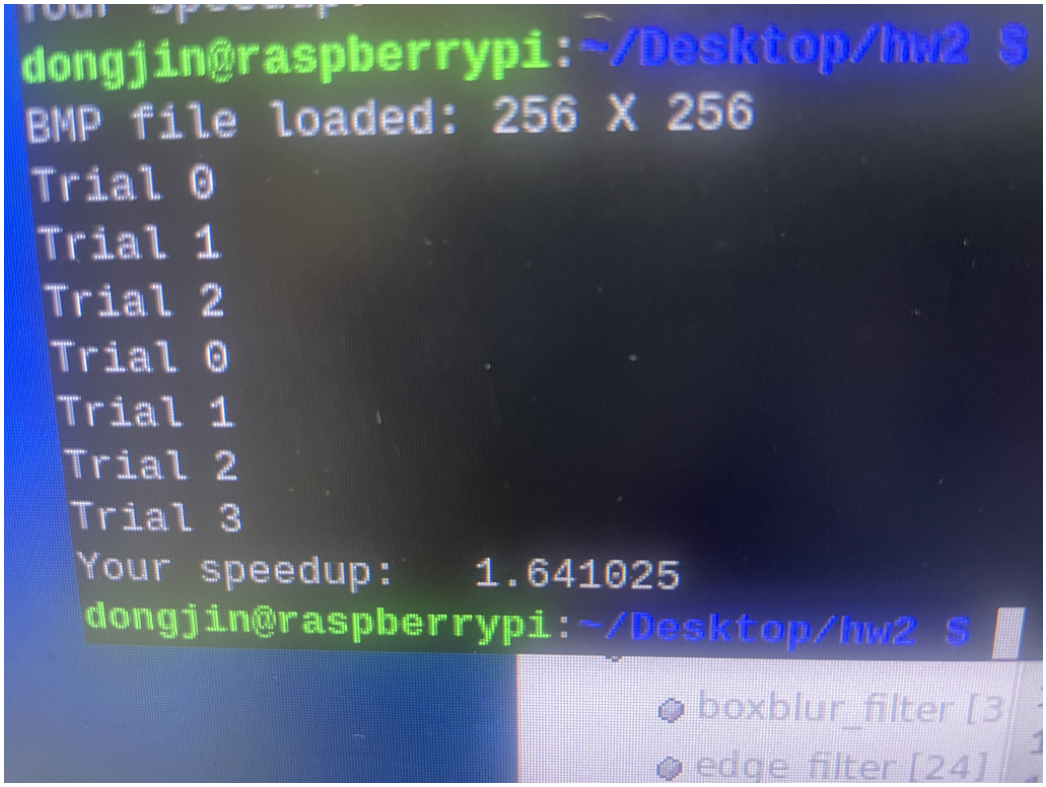
Separating the above boundary treatment reduced repetitive condition check calculations, saving CPU cycles. Therefore, I think this can also act as an optimization factor.

2. Result

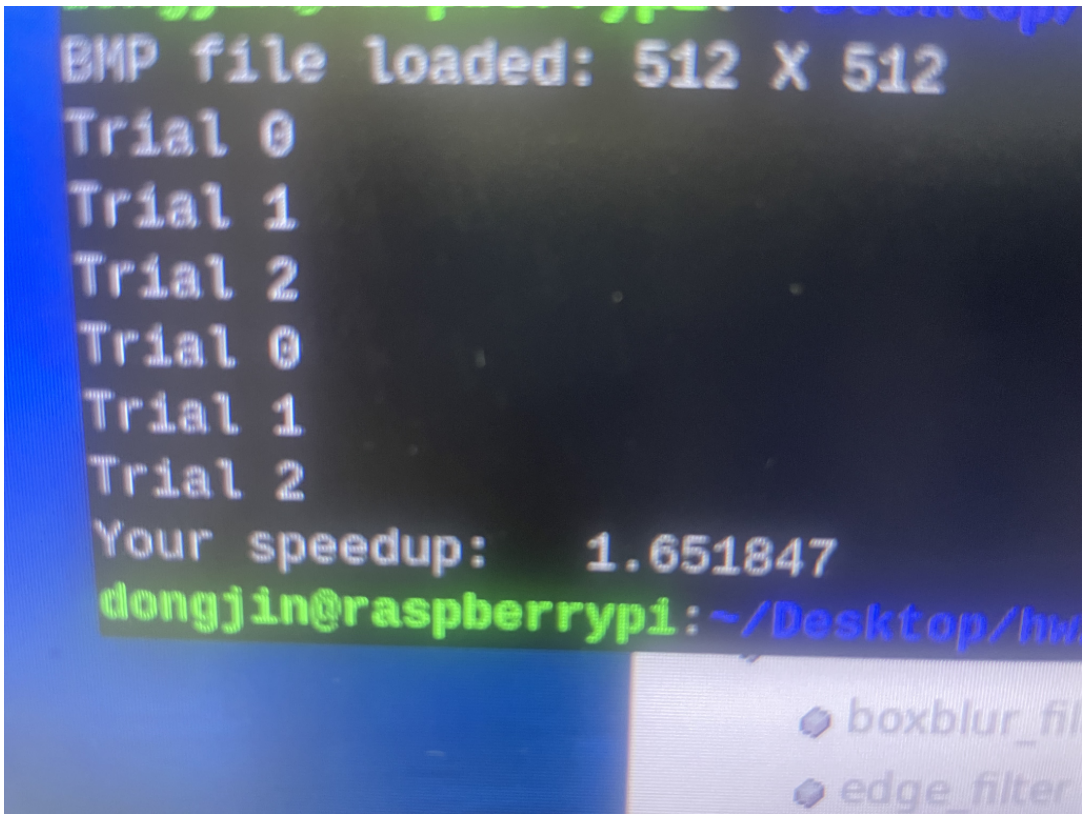
(1) img_128.bmp



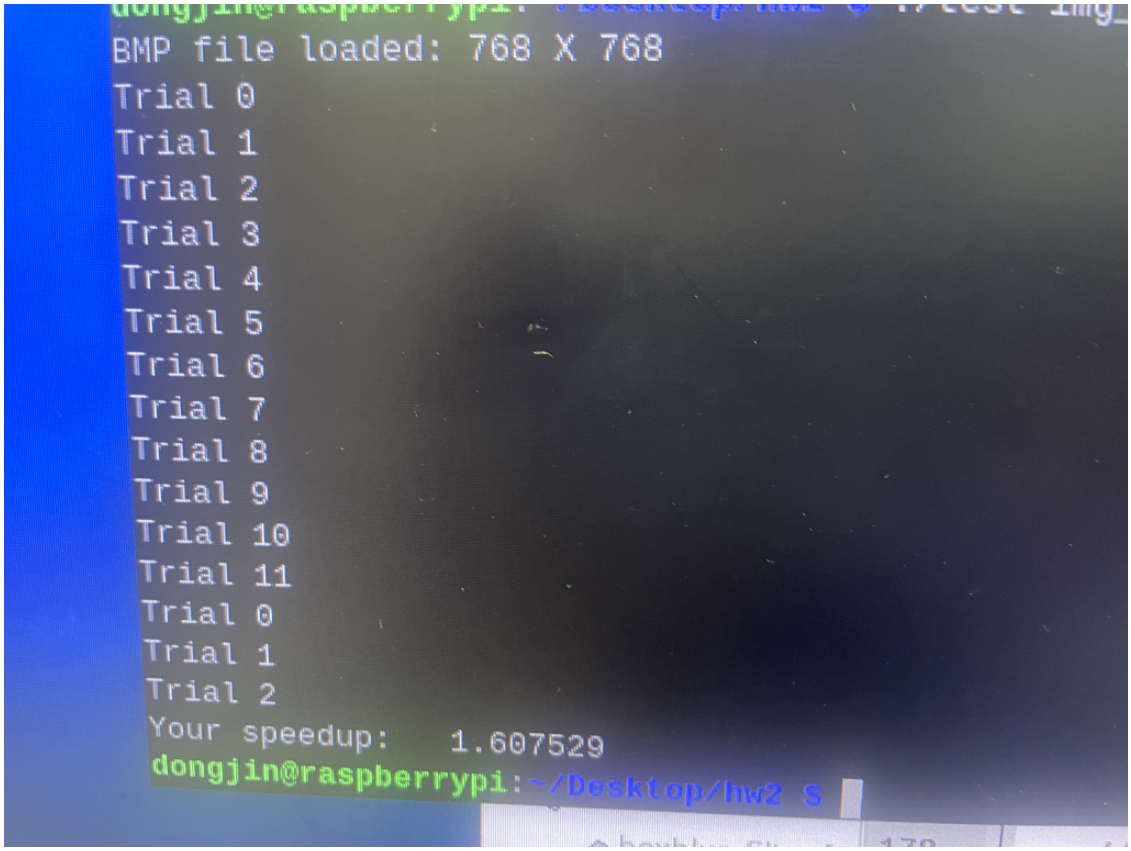
(2) img_256.bmp



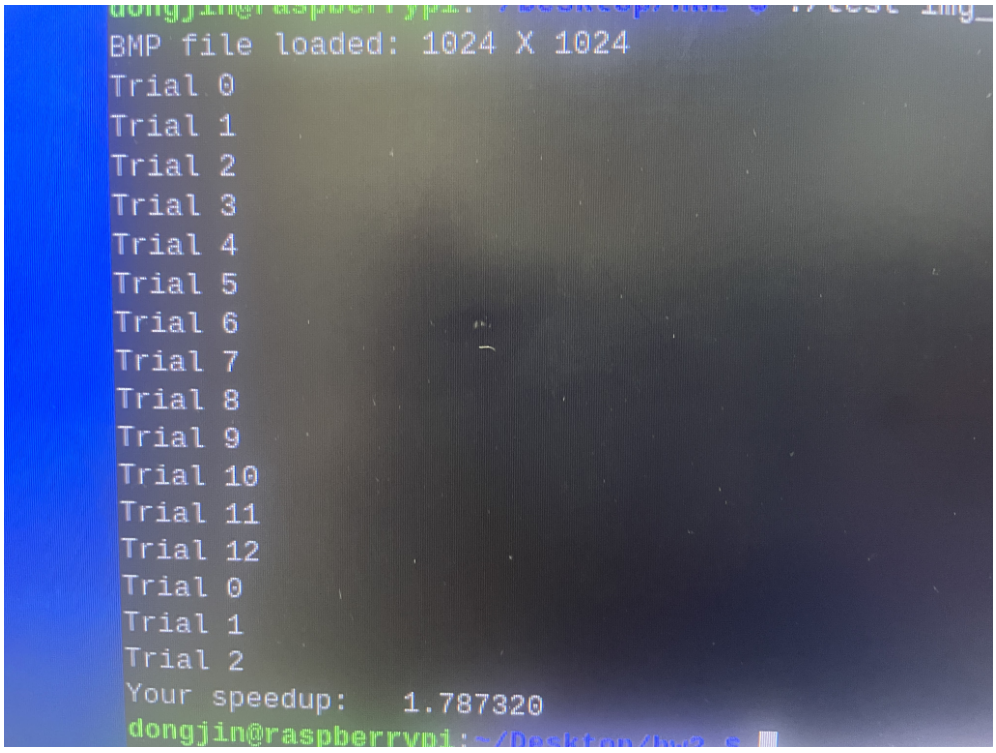
(3) img_512.bmp



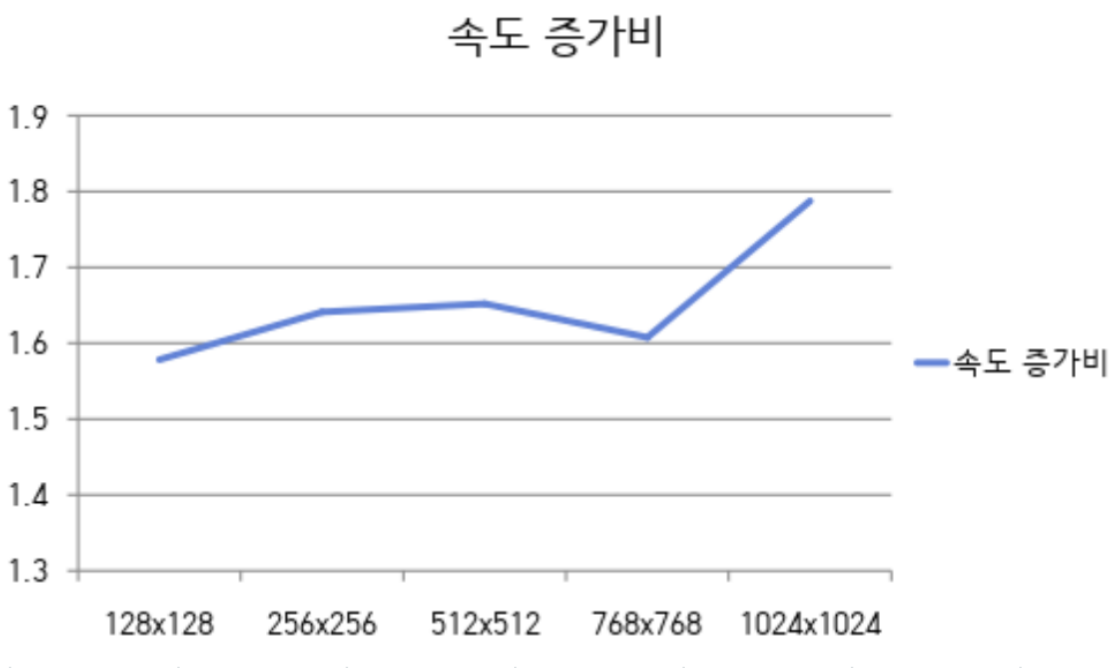
(4) img_768.bmp



(5) img_1024.bmp



Img SIze	128x128	256x256	512x512	768x768	1024x1024
SpeedUp	1.578262	1.641025	1.651847	1.607529	1.787320



Analysis

- It can be seen that as the size increases, most of the speed increase ratio increases slightly.
- The absolute time increased as the size increased, but the relative speed was faster than the original.
- On average, it can be seen that it is 1.6531966 times SpeedUp compared to the original.

conclusion

Two points of interest in optimization were cited: 'Removing unnecessary dynamic memory allocation' and 'Boundary processing and loop optimization'. The optimized code was implemented in these two parts, and it was confirmed that the average performance was about 1.65 times better than the original code.