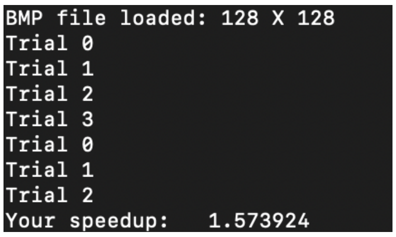
System Programming HW2 report

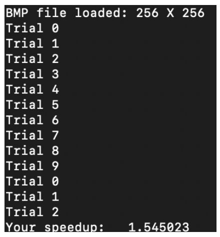
202211208 최종현

1. Implementation results

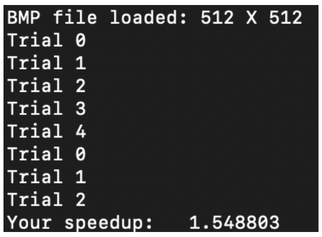
Img_size	128 x 128	256 x 256	512 x 512	768 x 768	1024 x 1024
speedup	1.573924	1.545023	1.548803	1.548866	1.913024



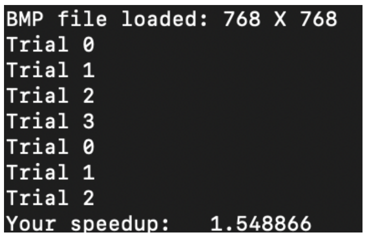
```
BMP file loaded: 128 X 128
Trial 0
Trial 1
Trial 2
Trial 3
Trial 0
Trial 1
Trial 2
Your speedup:   1.573924
```



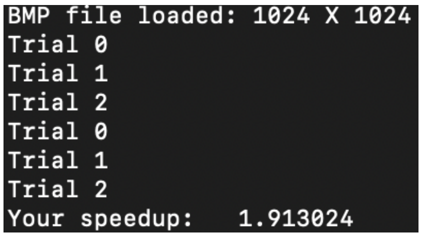
```
BMP file loaded: 256 X 256
Trial 0
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Trial 6
Trial 7
Trial 8
Trial 9
Trial 0
Trial 1
Trial 2
Your speedup:   1.545023
```



```
BMP file loaded: 512 X 512
Trial 0
Trial 1
Trial 2
Trial 3
Trial 4
Trial 0
Trial 1
Trial 2
Your speedup:   1.548803
```



```
BMP file loaded: 768 X 768
Trial 0
Trial 1
Trial 2
Trial 3
Trial 0
Trial 1
Trial 2
Your speedup:   1.548866
```



```
BMP file loaded: 1024 X 1024
Trial 0
Trial 1
Trial 2
Trial 0
Trial 1
Trial 2
Your speedup:   1.913024
```

2. Optimization approach

The performance issues in the original code primarily stemmed from unnecessary memory allocations and repeated conditional checks. In the initial code, memory was dynamically allocated and freed for each pixel using malloc and free, which resulted in significant overhead and performance degradation. To address this, dynamic memory allocation was removed, and necessary temporary variables were allocated on the stack, reducing the overhead associated with memory allocation and deallocation.

Next, the filtering operations involved numerous redundant conditional checks within loops, leading to unnecessary computations. Code motion was applied to optimize this by moving invariant computations, such as the calculations for max_x and max_y, outside the loops. This change alone resulted in approximately a 0.4x improvement in performance for 1024x1024 images.

Subsequently, loop unrolling was applied to the filtering operations to minimize loop overhead. For the 3x3 filtering operation, each pixel required nine computations, so manual loop unrolling was performed to handle each computation directly. This further reduced loop overhead and enhanced performance by approximately 0.2x.

Pointer arithmetic was then employed to optimize array access. Instead of using index-based array access, which is slower, pointers were used to access memory directly, improving memory access speed. This optimization was particularly beneficial for large images, where the impact on performance was more pronounced. By using pointers, cache hit rates were increased, and memory access times were reduced.

Finally, loop tiling was introduced to improve cache efficiency. Tiling helps to reduce cache misses and optimize memory access patterns by processing the image in blocks. The tile_size was adjusted to process the image in tiles, thereby maximizing cache efficiency. While the direct impact of this optimization was harder to measure, it is estimated to have contributed to an additional 0.2x improvement in performance.

By sequentially applying these optimization techniques, the performance of the filtering operation was significantly improved. The optimized code reduces loop overhead, optimizes memory access, and enhances cache efficiency, resulting in overall performance gains. However, it was observed that for images smaller than 1024x1024, the performance improvements were less noticeable due to the smaller data set size. As the image size increases, the speedup becomes more evident, indicating that the optimization techniques are more effective for larger images.