HW2

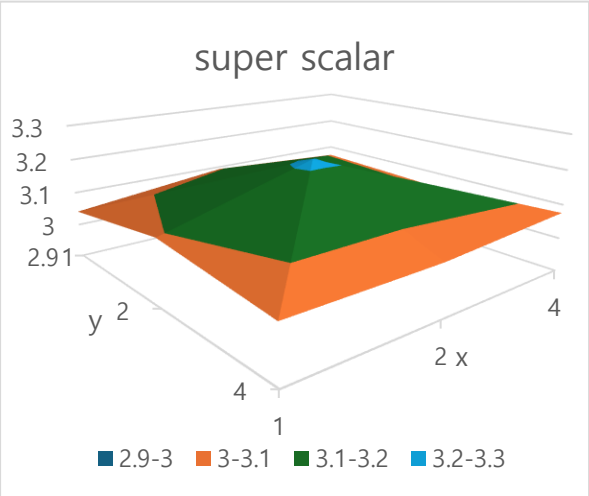
202011125 유선우

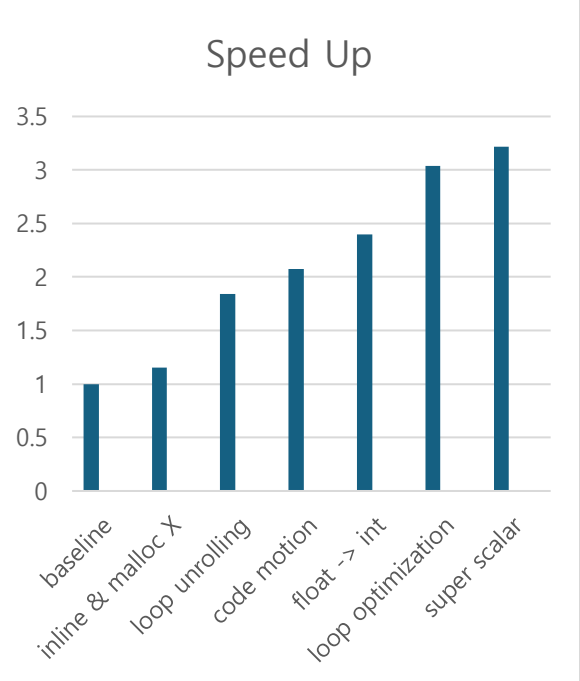
I modified the functions in the hw2.c file to optimize the convolution operation. The attempts made for this optimization are as follows. These attempts, as mentioned in the table of contents, were tested by executing them five times each on img_1024.bmp, and the average speed improvement over the baseline convolution operation was measured.

1. inline
2. malloc remove
3. loop unrolling (loop, if문 최적화)
4. code motion
   A. filter
   B. x + y * width
5. float -> int
   A. filter
   B. r, g, b
6. loop change (super scalar)
   A. various case

1. Section 1 (Implementation Result)

```
Your speedup:    3.661428
```





	avg	1st	2nd	3rd	4th	5th
baseline	0.9989418	0.996070	0.990056	1.021894	0.998636	0.988053
Inline, remove malloc	1.150373	1.157313	1.162038	1.143849	1.137740	1.150925
Loop unrolling	1.8441672	1.853397	1.776054	1.862501	1.823236	1.905648
Code motion	2.0756124	2.095205	2.068193	2.069813	2.075306	2.069545
float -> int	2.3960638	2.457367	2.362708	2.387391	2.355763	2.417090

2. Section 2 (Optimization Approach)

1. Baseline code
2. Inline, remove malloc (Improved)

By applying ‘inline’ to the Convolution and filter_optimized functions, faster function execution was enabled. Additionally, removing ‘malloc’ which supports dynamic allocation, allowed for faster procedure execution.

3. Loop Unrolling (Improved)

Through loop unrolling, the operations between the 3 * 3 filter and the image were executed sequentially. Additionally, instead of the previously used if-statement method, conditions were set assuming the filter size is 3 * 3 to execute the code. For example, in the situation where x > 0 && y > 0, the value of filter[0] can be multiplied by input[x + y*width].

4. Code motion (Improved)

Although the computation speed significantly improved with the above methods, code motion was used to further reduce repetitive operations. Specifically, to avoid repetitive memory access and multiplications, the nine values of the filter were each assigned to float variables. These filter variables were then used in the loop-unrolled scenarios as needed. Additionally, to reduce the multiplication involved in the repeatedly used x + y * width expression, the value was precomputed and assigned to a variable named xy_width for use in the code.

5. float -> int (Improved)

The variables for each filter value and the r, g, b values, which were previously used as floats, were changed to int data types. Additionally, during the convolution operations in the loop unrolling process, the p->r, p->g, and p->b values were cast to int when multiplied by the filter values. This change was made to improve speed, even if it resulted in a slight loss of accuracy.

6. Loop Change (Improved)

In the loop of the filter_optimized function, the previous process of assigning each r, g, b value separately was optimized by directly assigning the return value of the convolution function, thereby minimizing memory access. Additionally, since the loop used in filter_optimized varies in length depending on the image size, loop unrolling was not possible for this part. However, this loop was optimized using superscalar techniques to enable faster looping. The increments for the x and y values in each loop iteration were tested with values [1, 2, 4], and the performance was averaged over 5 runs to determine the optimal increment.
(The xy_width variable was used in this process to minimize the computational steps.)

(Rows correspond to x values, and columns correspond to y values.)

x \ y	1	2	4
1	3.0408492	3.0928766	3.0689744
2	3.093913	3.2164314	3.05391
4	3.0546114	3.0532026	3.0754986

While the exact reason is unclear, it was found that setting the strides of x and y to 2 resulted in the fastest computation speed. To further investigate this parameter, additional experiments were conducted. After several more trials, it was confirmed that a speed improvement of up to 3.661428 times could be achieved with this configuration.

```
Your speedup:   3.661428
```

As seen in the graph, the speed improvement is mainly significant in loop optimization. Using a large image of size 1024, which involves optimizing many loops, further enhances the performance improvement.