Name: 김나혜

Student ID: 202211017

<Implementation Results>

The following table shows the speedup of the optimized implementation (filter_optimized) compared to the baseline implementation (filter_baseline) for various image sizes:

pixel	try 1 speedup	try 2 speedup	try 3 speedup	Average speedup
128 X 128	1.521267	1.677021	1.658704	1.618997
256 X 256	1.814400	1.777193	1.699418	1.76367
512 X 512	2.018969	1.617057	1.401493	1.679173
768 X 768	2.431144	2.458691	2.403515	2.431116
1024 X 1024	2.357219	2.451584	2.495174	2.434659

The table indicates that the optimized function provides significant speedup across different image sizes, with speedups ranging from 0.5x to 2.4x. The speedup tends to increase as the image size increases, which is discussed further in the <Analysis of results from my optimization> section.

<Optimization Approach>

1. Loop Unrolling

Original Code: No loop unrolling applied.

Optimized Code: Loop unrolling was applied in the filter_optimized function to process four pixels at a time, reducing the number of iterations and improving performance.

2. Int Operations

Original Code: Used floating-point operations.

Optimized Code: Converted floating-point filter values to integers to speed up calculations.

3. Shift Operations Instead of Multiplication/Division

Original Code: Used multiplication and division.

Optimized Code: Used shift operations for normalization, replacing multiplication and division with shifts to save CPU cycles.

4. Minimized Pointer Usage

Original Code: Used pointers extensively inside functions.

Optimized Code: Reduced pointer usage by storing intermediate results in local variables.

5. Memory Aliasing Consideration

Original Code: Each memory access was recalculated in the loop, leading to inefficient memory usage.

Optimized Code: Precomputed row and column indices to minimize repeated calculations and improve cache efficiency.

6. Function Call Optimization

Original Code: Called convolution function repetitively inside nested loops.

Optimized Code: Reduced the number of function calls by unrolling loops and processing multiple pixels in each iteration.

7. Improved Cache Efficiency

Original Code: Used malloc and free inside nested loops, which could lead to cache inefficiencies.

Optimized Code: Removed dynamic memory allocation (malloc and free) from the inner loop to improve cache performance.

<Discuss of results from my optimization >

Reason for Increased Speedup with Larger Images

- Reduced Function Call Overhead: In smaller images, function call overhead is relatively significant. However, as image size increases, the function call overhead becomes a smaller fraction of the overall execution time, resulting in better speedup.
- Effective Loop Unrolling: Loop unrolling techniques yield greater performance improvements with larger images due to the increased number of iterations.
- Better Cache Utilization: Optimized memory access patterns and reduced dynamic memory allocation improve cache performance, which is more noticeable with larger images.