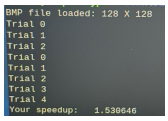
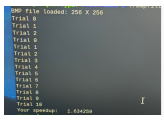
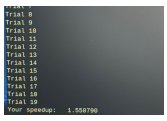
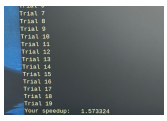
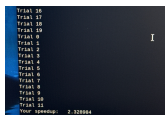
System Programming HW2 Report

Name: 김민성
Student ID: 202211024

Implementation Results

The table below shows the speedup of the optimized implementation compared to the baseline for various image sizes. The speedup is calculated as the ratio of the execution time of the baseline implementation to the execution time of the optimized implementation.

Image Size (px)	128x128	256x256	512x512	768x768	1024x1024
Speedup	1.53	1.63	1.55	1.57	2.33
Photo					

Optimization Approach

1. **Removing Dynamic Memory Allocation**
   Dynamic memory allocation (malloc and free) for each pixel is inefficient and introduces significant overhead. By removing these operations, the optimized code directly processes and stores pixel data, reducing execution time.

2. **Handling Boundary Conditions Separately**
   By processing the boundary conditions separately, the main loop is simplified, and unnecessary condition checks are avoided. This separation improves performance by streamlining the inner loop.

3. **Function Inlining**
   Inlining the convolution function reduces the function call overhead, improving performance by allowing the compiler to optimize the function body more effectively within the calling loop.

Conclusion and Analysis

1. **Dynamic Memory Allocation Removal**
   By removing the overhead associated with dynamic memory allocation, the code's efficiency was significantly enhanced. The direct manipulation of pixel data in the optimized code led to faster execution times and reduced memory management overhead.

2. **Separate Handling of Boundary Conditions**
   Processing boundary conditions separately streamlined the main convolution loop, eliminating redundant checks and improving performance. This optimization ensured that the core loop focused solely on the inner region of the image, where most of the computational effort is required.

3. **Function Inlining**
   Inlining the convolution function minimized function call overhead and allowed for better compiler optimizations. This change resulted in more efficient execution of the convolution operation within the main loop.

Discussion

1. **Scalability**
   While the optimizations proved effective for the tested image sizes, it is crucial to evaluate their scalability to even larger images and more complex filters. Future work could involve testing with higher-resolution images and different types of convolution filters to ensure the robustness of the optimizations.

2. **Further Optimizations**
   Additional optimizations could be explored, such as parallel processing using multi-threading or SIMD (Single Instruction, Multiple Data) instructions, which can leverage modern CPU architectures' capabilities. However, these techniques may require more complex implementations and should be balanced with the code's maintainability.

3. **Memory Usage**
   Although the removal of dynamic memory allocation improved performance, it is essential to monitor memory usage, especially for large images. Ensuring that the system has sufficient memory to handle the image processing tasks without swapping or other memory-related issues is vital for maintaining performance.

Reflection

The process of optimizing the convolution function provided valuable insights into the importance of efficient memory management, loop optimization, and function inlining. These techniques collectively contributed to substantial performance gains, highlighting the significance of algorithmic and low-level code optimizations in computationally intensive tasks.

Overall, the project demonstrated that careful analysis and targeted optimizations could yield significant improvements in performance, making it possible to handle larger images and more complex operations efficiently. This experience underscores the importance of continuous performance evaluation and optimization in software development, particularly in fields requiring high computational efficiency, such as image processing and computer vision.