1. Implementation Results

The table below shows the speedup of the optimized implementation (`filter\_optimized`) compared to the baseline implementation (`filter\_baseline`) for various image sizes.

|  |  |
| --- | --- |
| Image Size | Speedup |
| 128x128 | 1.323 |
| 256x256 | 1.306 |
| 512x512 | 1.333 |
| 768x768 | 1.330 |
| 1024x1024 | 1.329 |

2. Optimization Approach

1) Memory Access Pattern Optimization

- I tried to make the memory access pattern better. By accessing memory in a sequential way, I made the cache hit rate higher. This helped to reduce the time taken for memory operations.

- This strategy alone gave a significant speedup. When memory is accessed sequentially, the data needed for processing is likely already in the cache, reducing the number of cache misses.

2) Eliminating Redundant Memory Allocations:

- In the original code, there were too many memory allocations and deallocations for each pixel operation. I removed these redundant operations and used a single output buffer instead. This reduced the overhead associated with memory management.

- This strategy improved performance by reducing the time spent in memory allocation and deallocation functions.

3) Loop Unrolling:

- I applied loop unrolling to reduce the overhead of loop control and increase instruction-level parallelism. This involved manually expanding the loop body to perform multiple operations in each iteration.

- Loop unrolling provided a modest speedup by reducing the number of iterations and the associated loop control instructions.

The combination of the above strategies resulted in speedups for different image sizes. These results indicate that the combined optimization strategies consistently improved performance across different image sizes.