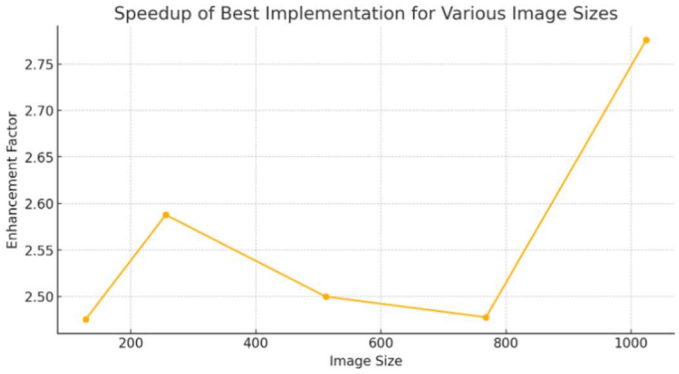
System programming: hw2_optimization

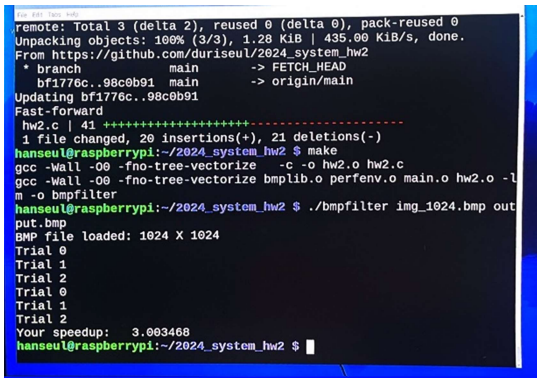
202211211 최한슬

1.Implementation results:

Size of img	128	256	512	768	1024
	Min	Min	Min	Min	Min
	2.346916	2.446939	2.389771	0.846469	2.651536
	Max	Max	Max	Max	Max
	7.083084	2.694658	2.701009	2.569501	3.003468
	*Average	Average	Average	*Average	Average
	2.475343	2.587811	2.499771	2.477668	2.77592

*Average of 9 values (except a edge case) / other averages are calculated with 10 values





2.Optimization approach

Original Implementation

The original code performed a convolution operation applying a 3x3 filter using nested loops. The main inefficiencies were as follows: • Using double-precision floating-point (double) for intermediate calculations. • Using dynamic memory allocation (malloc and free) for each pixel operation. • Conditional checks for boundary conditions. • Frequent cache misses due to inefficient memory access patterns.

Optimization Strategies

1. **Inlining and Float Usage**

   • Defined the convolution function as static inline to encourage inlining and reduce function call overhead. • Changed intermediate calculations from double to float to reduce memory usage and computation time.

2. **Pre-calculation of Indices**

   • Pre-calculated frequently used indices (x - 1, x + 1) to avoid repeated calculations.

3. **Loop Unrolling**

• Unrolled the loops for dy and dx to reduce loop control overhead and increase instruction-level parallelism.

4. **Efficient Boundary Checking**

   • Moved boundary condition checks outside the innermost loop.

5. **Cache-Friendly Memory Access**

   • Improved memory access patterns by using pointers (input_num and input_ptr) to access pixel data linearly.

6. **Clamping with Ternary Operator**

   • Used the ternary operator to clamp RGB values between 0 and 255.

7. **Pre-calculation of Filter Values**

   • Pre-calculated filter values into a local array for quick access during the convolution operation.

8. **Removal of Dynamic Memory Allocation**

   • Removed dynamic memory allocation (malloc and free) within the loop and wrote results directly to the output array.

9. **Batch Processing**

   • Modified the filter_optimized function to process two pixels at a time.

Results The optimized implementation significantly improved execution speed and efficiency. The main benefits were: • Reduced computational overhead by decreasing the number of arithmetic operations and conditional checks. • Improved cache performance due to better memory access patterns. • Enhanced overall performance by eliminating dynamic memory allocation.

Failed strategy

In the filter_optimized function, I attempted to precompute the filter to improve memory access patterns by replacing direct access to the filter array in the convolution_optimized call with a precomputed_filter.

```
float precomputed_filter[9] = { 0 };
for (int i = 0; i < 9; ++i) {
    precomputed_filter[i] = filter[i];
}
```

However, this change reduced performance enhancement. The precomputation overhead for computing and storing these filter values, which did not outweigh the theoretical benefits of faster access during the convolution operation. Even though I unrolled them into 9 lines, still performance reduced. Additionally, storing precomputed values negatively impacted the cache efficiency, leading to slower overall performance despite the optimized access patterns.