Assignment 2: Efficient Program Implementation

202211080 박시윤

1. Implementation results

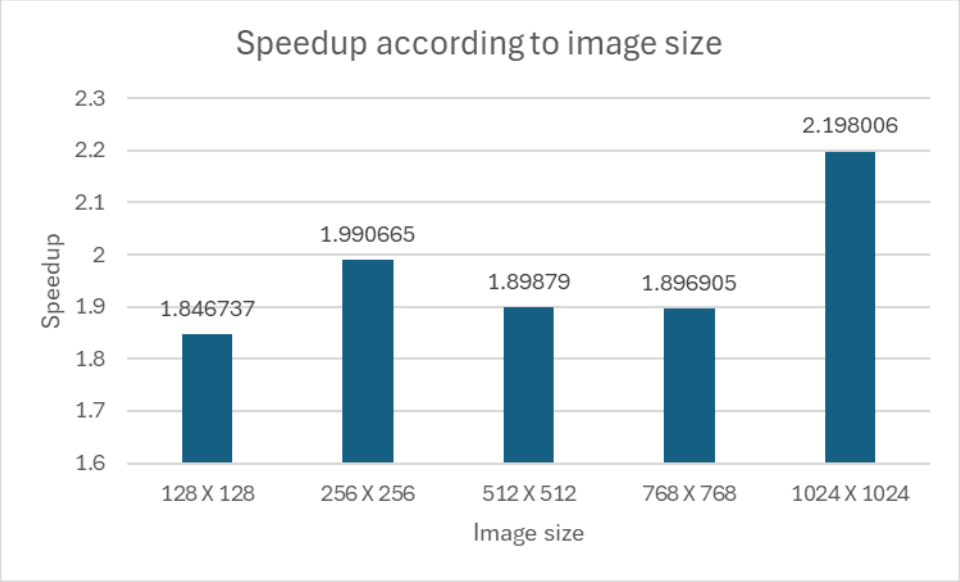

Fig. 1. Speedup according to image size

Using the code with the highest speedup, I checked how the speedup varied depending on the image size. When the image size was 1024 X 1024, the speedup was highest at 2.198006, and there was no clear trend for image sizes smaller than this. This may be because the code execution time is shortened due to the small image size, resulting in an error. Therefore, in the optimization approach section, analysis was conducted using a BMP file with a size of 1024 X 1024.

2. Optimization approach

1) Dynamic memory allocation removed

Compared to the provided code, the ‘Dynamic memory allocation removed’ code, which eliminated dynamic memory allocation and used static memory allocation that allocates directly to the stack, showed a speedup of 1.391691. ‘malloc’ and ‘free’, which are used for dynamic memory allocation, have slow execution speeds and are inefficient when included in a nested for loop, such as provided code. As a result of changing the code to eliminate dynamic memory allocation and directly assign the ‘Pixel’ structure to the stack, the execution speed was 1.391691 times faster.

2) Code motion

In the code where ‘Code motion’ was applied, code that was included within the for loop and was executed a large number of times was moved outside the for loop. Since they always produce the same result regardless of repetition, the execution time of the code can be reduced by reducing the frequency of execution. Additionally, the inefficiencies caused by using unnecessary if statements multiple times were replaced with the ternary operator. The ‘Code motion’ code showed a speedup of

1.702522, which is the result of additionally applying code motion in the ‘Dynamic memory allocation removed’ code, so the actual speedup can be interpreted as approximately 1.310831.

3) Loop unrolling (inner loop)

In the ‘Loop unrolling (inner loop)’ code, the code was repeated as many times as the number of repetitions of the inner loop without using a for loop. At this time, the part containing the loop variable was modified to the final result calculated by directly substituting the loop variable value. As a result, one for loop was replaced with three repetitions, and the execution speed was increased by 1.891324 times. Since the ‘Loop unrolling (inner loop)’ code is a code that applies additional loop unrolling to the ‘Code motion’ code, the actual speedup can be interpreted as approximately 1.188802.

4) Loop unrolling (outer loop)

In the ‘Loop unrolling (outer loop)’ code, the outer loop is unrolled in the same way as ‘Loop unrolling (inner loop)’. As a result, a speedup of 2.198006 was achieved. The ‘Loop unrolling (outer loop)’ code is a code that applies additional loop unrolling to the ‘Loop unrolling (inner loop)’ code. Therefore, the speedup achieved by outer loop unrolling is 1.306682, and the speedup achieved by total loop unrolling is 1.495484.

5) Approaches that fail to speedup

In addition to the above-described approaches, we also tried changing the multiplication operation into multiple addition operations and moving the variable declaration inside the for loop to outside the for loop. The speedup was 1.653080 and 1.662310, respectively. Since this is applied to the ‘Code motion’ code that achieved a speedup of 1.702522, it can be interpreted that the execution speed has actually decreased. Therefore, this method was not applied to the final code.

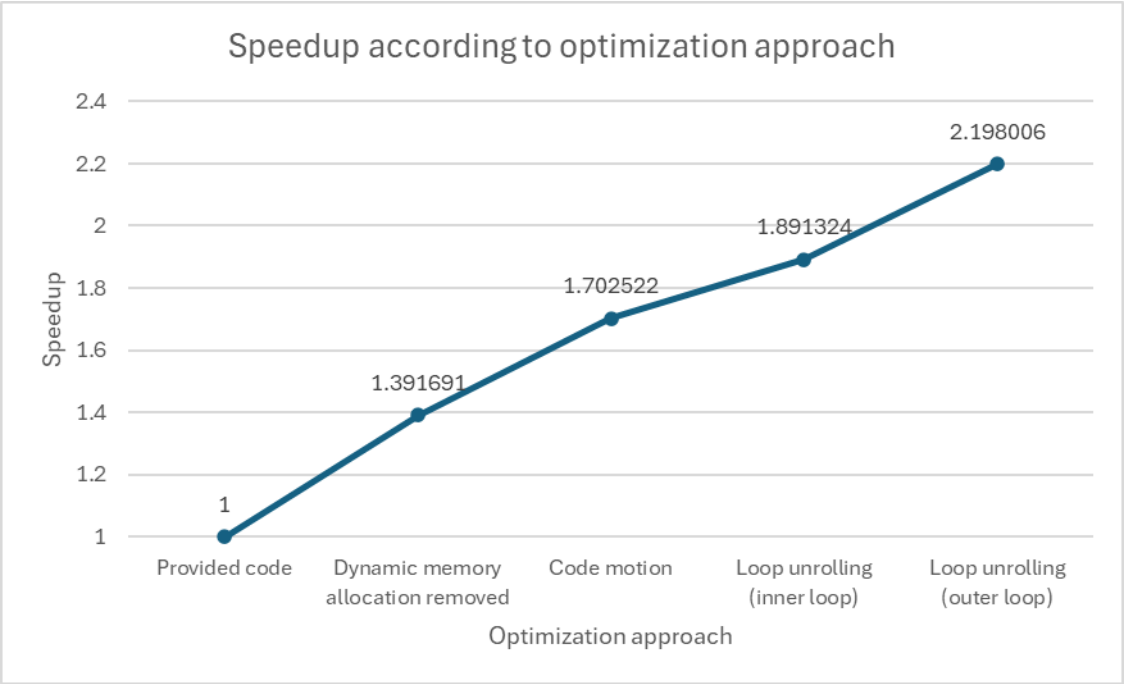

Fig. 2. Speedup according to optimization approach