Assignment 2: Efficient Program Implementation

202211002 강규영

1. Implementation results

	Speedup1	Speedup2	Speedup3	Speedup4	Speedup5	Average speedup
Img_128	3.678	3.649	3.867	3.680	3.599	3.695
Img_256	3.725	3.964	3.735	3.722	3.747	3.779
Img_512	3.759	3.756	3.788	3.778	3.771	3.770
Img_768	3.770	3.833	3.787	3.800	3.806	3.799
Img_1024	4.347	4.437	4.202	4.309	4.306	4.320

It is a table that measures and averages the speedup 5 times for a total of 5 sample images. When looking at the average speedup, it was confirmed that the higher the size of the image, the higher the speedup, because the total capacity decreases when the sample is small, so most of the data goes into the L1cache and becomes faster. When the image reaches 1024, seeing that the average speedup exceeds 4, it can be expected that there will be more significant results in large-scale data.

2. Optimization approach

2-1. First, in the filter_optimized function, there was an intermediate step involving malloc before assigning values to output. This step was removed, and the process of assigning values to output was delegated to the convolution function.

2-2. In the convolution function, loop unrolling was implemented, which involves reading filter values and input values multiple times directly instead of using a for loop. This significantly improved performance.

2-3. In the convolution function, rather than reading values from memory and using them only once, the convolution operation was modified to process 16 values at a time (from (y, x) to (y+1, x+7)). This allowed for up to six uses per memory access, significantly speeding up the process. Since the convolution function handles the allocation to output, there was no need for an additional buffer or memset. Given that each pixel of the input image is 3 bytes, and assuming a cache block size of 64 bytes, approximately 21 pixels can fit in the cache. Rather than doing 16 in order, the performance was improved by dividing it into 8 pixels.(y, y+1 each 8 pixels) Additionally, since the width is fixed at a multiple of 32, this approach posed no problems.

2-4. The main file includes code that tolerates up to 10 differing values, so integers (int) were used instead of floats (double,float) to accelerate computations, which resulted in performance improvements without issues. Although both float and integer functional units exist, performance was better when using only integers rather than splitting the operations between both units.

2-5. Instead of accumulating multiplication results into a single variable, three variables were used alternately, and their final sums were combined at the end. This approach was faster when using one variable, but there was no significant difference when using two or three variables. Additionally, a second variable replicating filter values was utilized concurrently.

2-6. Given that the L1 cache size is 32KB, and the image size is 3MB for a 1024x1024 image, it was determined that the entire data set could not fit into the cache. Therefore, convolution was attempted in blocks, with sizes such as 64x64 and 32x32. However, this approach resulted in lower speedup compared to sequential convolution. This might be due to implementation errors or the possibility that cache differences do not have a substantial impact.

2-7. Unnecessary conditional statements were removed or consolidated into single else if comparisons.

2-8. Convolution function was converted to void to eliminate returns, and inline was used, though this did not result in significant performance improvement.

2-9. I also tried to combine the convolution function with the filter_optimized function, but it didn't have much effect.