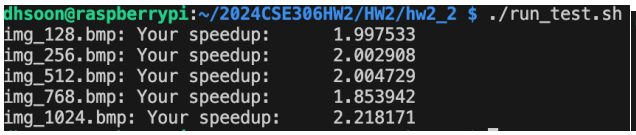
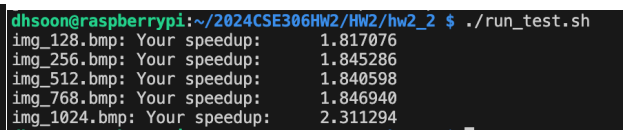
Report

202211102 순동현

1. Implementation results

1.1 Compare using different image sizes (using shell files)





1.2 Use a large image file(img_1024.bmp)

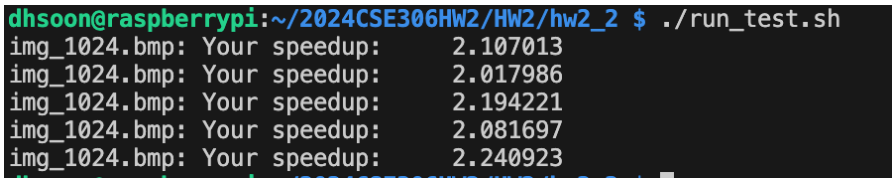


Image size	Img_128.bmp	Img_256.bmp	Img_512.bmp	Img_768.bmp	Img_1024.bmp
Speed up	1.997533	2.002908	2.004729	1.853942	**2.311294**

2. Optimization approach

The original HW2.C code was implemented by allocating and deallocating memory for each pixel each time, which resulted in high memory allocation/deallocation overhead and poor cache usage efficiency.

The optimized code in this HW2 introduces techniques such as cache blocking, loop unrolling, common partial expression elimination, and reassociated computation to improve performance by about 2.3x or more. I first divided the input image into smaller blocks to better fit into the CPU cache, reducing cache misses and improving data locality. I learned that reducing miss rate is the most important thing, so I increased locality, and this approach helped keep frequently accessed data in cache longer. I checked the L1 and L2 cache sizes by printing them out during the cache blocking process and set the pixel block size to 8*8, 16*16, and 64*64, and found that 32*32 gave the best results. I also used loop unrolling, a technique that reduces the loop's overhead and improves performance by reducing the number of

iterations of the loop. Unlike the original code, in my code I implemented loop unrolling by changing the innermost loop to process four pixels at a time. This allowed me to reduce the number of iterations of the loop and use the CPU pipeline more efficiently. Before applying loop unrolling, the innermost loop that applies the filter for each pixel processed one pixel each time, repeatedly calculating the index and processing each pixel value individually each time the filter was applied, which was very expensive. So, while the average value of speedup before applying loop unrolling was low at 1.4-1.5, I was able to see a 1.7-1.8x improvement in performance after applying loop unrolling.

In addition, we used conditional statements to process additional pixels as long as they didn't exceed x_max. This allowed us to process multiple pixels at once to increase data locality and improve cache efficiency.

I also optimized my code by using reassociated computation. The original code repeatedly multiplies and sums the filter and input image data to calculate the RGB value of each pixel. This process repeatedly calculates the filter index and pixel index, which is very inefficient. In this process, we used the common subexpression elimination technique, which is implemented by pre-calculating and reusing the filter index and pixel index. I reduced the cost by calculating and reusing fewer base_indexes and pixel_indexes and using the results of these calculations to update the RGB values. I also changed the order of calculation to achieve the same result with fewer operations.

I tried to optimize further by vectorizing and some other methods, but I was not able to get any performance improvement. Also, I was not able to use multi-threading and SIMD (ARM NEON), so I was not able to use parallelism efficiently to get much performance. Through this challenge, I was able to realize the efficiency of multi-threading and parallelization.