SYSTEM PROGRAMMING HW2 REPORT

Student ID: 202011003

Name: 강승수

1. Implementation results

The following table shows the results of three measurements for each image size on Raspberry Pi.

Image size	1	2	3	Average speed up
128	2.30273	2.470721	2.265462	2.346304333
256	2.61616	2.383489	2.349989	2.449879333
512	2.462356	2.442651	2.411738	2.438915
768	2.417449	2.367482	2.421167	2.402032667
1024	2.800714	2.741307	2.766285	2.769435333

<table 1: speed up results by image size>

For Image size of 128, 256, 512 and 768 was around 2.4x. However, for an image size of 1024, the speed up was approximately 2.7x. Using smaller image files may result in execution times that are too short to accurately measure speed up. Testing with an image file size of 1024 yields more stable and representative speed up results.

2. Optimization Approach

First, let's discuss the 'convolution' function. This function applies a 3*3 convolution filter to the surrounding area of a specific pixel to calculate a new pixel value.

Boundary handling: The function takes precomputed input for whether x and y are boundary parts. Depending on this input, it executes different process. This distinction is made because boundary parts require many if statements, which can reduce speed up. Therefore, different processes are handled for boundary parts and non-boundary parts.

When Not a Boundary: We retrieve the values of the 9 surroundings pixels from the input. These 9 addresses are stored in p0 to p8. Accessing via addresses makes the code more compile friendly. Then, we sum the r, g, b values. There are no for loops in this process.

When a Boundary: Boundary parts require if statements for each cell. This process uses nested for loops to determine whether each cell is boundary, and only adds the r, g, b values if it's not a boundary. The focus was simplifying the calculation process of the innermost for loop. Frequently executed and invariant values were precomputed. The cache's array storage method was considered. Accessing the output via pointers mad the code more compile friendly.

Second, let’s discuss the filter optimized function. This function iterates over each pixel in the image, calling the ‘convolution’ function for each one. It receives several parameters and analyzes whether the pixel is a boundary within nested for loops, then inputs these parameters into the ‘convolution’ function. loop unrolling count of 3 were applied. Constants(“const”) were used for invariant variables to improve computational efficiency, and the innermost for loop was simplified as much as possible. Frequently executed and invariant values were precomputed. The cache’s array storage method was considered. Accessing the output via pointers mad the code compiler friendly.

Failed Strategies

< Convolution function>

I make two codes: one that changes the process based on whether it’s a boundary, and the one that always uses if statements regardless of boundaries. The former showed higher speed up, so I chose it.

When not a boundary, I implemented and optimized using nested for loops, but the result was lower than the current implementation

<filter optimized function>

I tested loop unrolling with counts of 1(no unrolling), 2, 3, 6 and 10. The speedups were 2.54 for 1, 2.68 for 2, 2.76 for 3, 2.63 for 6 and 2.45 for 10 on Raspberry Pi. Thus, I chose a loop unrolling count of 3.