System Programming Project
Program Optimization Report

201811061 Park Dongjin

1. Result

	128 x 128	128 x 128 (server)
speedup	1.143763	1.051651
	256 x 256	
speedup	1.271431	
	512 x 512	
speedup	1.364131	
	768 x 768	
speedup	1.427640	
	1024 x 1024	1024 x 1024 (server)
speedup	2.073155	2.982123

2. Optimization approach

(1) First step : single thread

- Code motion(in convolution function and filter_optimized function) :

  In conditional statement(if) x + dx and y + dy are calculated every time in for loop. To make x_accum = x + dx, y_accum = y + dy, calculation is out of for loop. It reduces frequency with which computation perfomed. And before entering dx loop, calculation Pixel address is done. Pixel* input_y= input + (y + dy)*width. By calculating address, computation frequency is reduced too. Before entering dx loop, calculation dy_accum = 3*dy + 4. So computation frequency is reduced, because filter address is dx + dy_accum. In filter_optimized function, before entering x loop, Pixel address is calculated. As input_y, it makes computation frequency reduced.

- Spatial locality(in convolution function and filter_optimized function) :

  The order of dx loop and dy loop was changed. so Pixel array(Pixel* input) and float array(float* filter) are accessed on a column-by-column basis. That makes spatial locality better. Because C arrays allocated in row-major order. In filter_optimized function, the order of x loop and y loop was changed too but after multi-thread step, it merged with the convolution function.

- Branch(in convolution function) :

  There are 6 IF in the original convolution function after the loop. To reduce branch instruction, else if statement is made. If r, g, b value are smaller than zero, cases greater than 255 do not need to be considered. So it removed unnecessary branches. And in dx loop, there are two IF statement but The if statement related to y can exit the dx loop. Because it is independent of dx. so it removed unnecessary branches.

- Inline function (in convolution function) :

  Convolution functions are used in filter_optimized functions. Using inline will help solve the optimization blocker(Procedure Calls).

- Data type (in convolution function) :

  R, g, b variables in the original convolution function are double type. Double type has 8 bytes. In dx loop, there is multiplication between unsigned char and float. The result of the multiplication is

float. In the C language, if you mix data types, the compiler performs implicit type conversion, and the data type is automatically converted to a larger size and wider expression range. So result of r, g, and b can have float type. After type change(double -> float), speedup is increased. Because float type has 4 bytes. It's smaller than double type.

- Speedup after First Step(single thread) :

After applying all the optimizations written above, the speedup of 1024 x 1024 images on the server was about 1.5. To further increase speedup, multithread programming was used. Since r, g, and b are calculated independently, the speedup is expected to increase if multi-threading is used.

(2) Second step : multi thread

- pthread_create (in filter_optimized function) :

To use pthread, <pthread.h> is included. After creating a thread for each of r, g, and b, pass the convolution function and an args array as arguments.

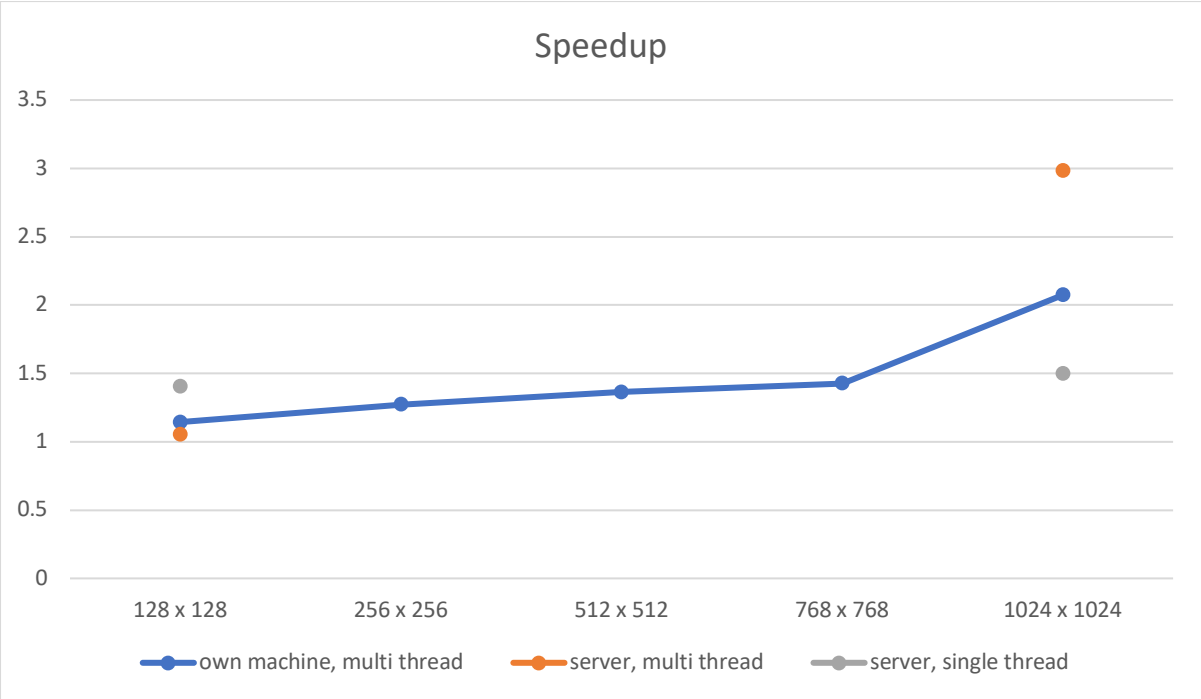
- convolution_r, g, b :

The width, height, input, output, and filter are assigned from args. The x loop and y loop in the original filter function were combined with the convolution function. All of the code optimization techniques previously applied were used while merging. So a total of 4 multiple loops were created. Like the existing convolution function, the process of summing r, g, and b is in the dx loop. After dy loop ends, r, g, and b is updated using the output address. Then, r, g, and b are initialized to 0, and the following x loop is performed.

- pthread_join (in filter_optimized function) :

Use the join function to wait for each thread to exit. When all threads are terminated, the filter function is terminated.

3. Result analysis



-Speedup Table in single thread (server)

	128 x 128(server)	1024 x 1024(server)
speedup	1.402930	1.498020

According to the instructions provided, the speedup of image files that are too small cannot be completely measured. It can be seen that the speedup increases as the size of the image increases. From this, it can be seen that multi-threading is more efficient as the image size increases. Moreover, the processor of the own machine (VirtualBox) is dual-core, and the server has eight cores. Because the speedup on the server is faster, you can see that multi-threading is more effective as the processor has more cores. When single-threaded, the speedup of 128 images(on server) was about 1.4. However, when 128 images were operated in multi-thread(on server), speedup decreased. This suggests that the overhead of creating threads is relatively increased compared to when the image was large. In single-thread, the improvement of speedup is smaller than that of multi-thread when the image is large. Since optimizing a large image is more efficient than optimizing a small image, using multi-threading is more effective than single-threading.

4. Execution precaution

When running the code on the server, there is a compiler warning, but the result worked fine. However, when there is a warning, the process may be terminated intermittently, so please run the code several times.