System programming Project

201911050 도담록

1.Methods

1-1. Basic Methods

First, I declared all iterators out of for-loop, preventing repeated declaration. Also, I defined every function Inline, removing the time it takes for the function to be called. But there is huge function, convolution_block, so a compiled file may be too big. But in this project file size is not a problem, I decided to use all function inline function.

At the Function Filter_optimized, I changed the order of x for loop and y for loop to use locality, Because Input and output are connected by x, not y. And I declared x+y*width out of x for loop, preventing doing same calculation.

After doing convolution calculation, RGB value may exceed 0-255 range. This can make overflow, because our pixel value is unsigned char, which has 0-255 range. So, this program should check if RGB value exceed or not and modify them. To treat with this problem and make this sequence optimized, I used two method. First method is: check if RGB value is 0-255. If true, change RGB float to unsigned char directly. Else, Check if RGB value exceed 0 or 255 and make them 0 or 255. Second method is to make if else if. If R exceeds 255, R must not exceed 0! But using first method, there might be small difference with basic code cause of order change, float might fail to fractional operations. In main.c, boxblur_filter can make this small error. So, there is an error of about 1 per 100 pixels. Also, I used “start” instead of “x, y” to make convolution function does not calculate index value.

In convolution function, I eliminated every for loop, removing every useless iterator calculation. It was possible because count of loop is so small (x loop 3, y loop 3). And I declared values used repeatedly, like filter values, which are used 3 times each (R, G, B). And I dismantled as it was, so I could use same method in Function filter_optimized; Change values when (Virtual)y changes.

When I put calculated results in the output, I used references instead of calling directly, so I put a value by out->r=(value), out->g=(value), out->b=(value). And convolution_in function returns array with length 3, which is [r, g, b] instead of using Pixel structure. Also, convolution_block function returns none and put a value directly to output.

1-2. Convolution_in and 4*4 Convolution_block

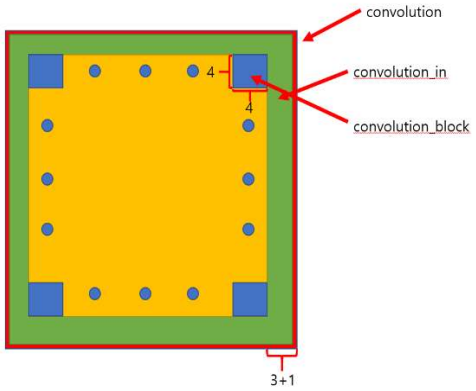

figure 1_convolution functions in proj.c

There are three convolution function in my proj.c. First is convolution, which is used just for edge and corner, to make every case safe. Second is convolution_in, which is used for inner 3 lines. And for the other points I used convolution_block, which is main function of my project, which calculates 4*4-pixel convolutions at once. It was possible because input data can be split with 32*32 matrix. So, if image is big enough, the degree of optimization mainly depends on the optimization of convolution_block.

Convolution function is basic function. In this function I used methods in 1-1. Convolution_in function is a function which uses that points are not in edge and corner, so there are no if code for exception handling. These two functions had locality, but by breaking the function into three, the locality of the vertical axis points decreased, but it is small compared with the benefits. Also, not in proj.c, I used unroll (factor=8) for Convolution_in for loop, increased speed little but I deleted it because convolution_in is not main function.

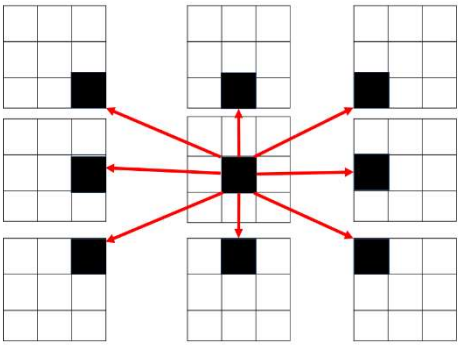

Figure 2_ pixel which called 9 times

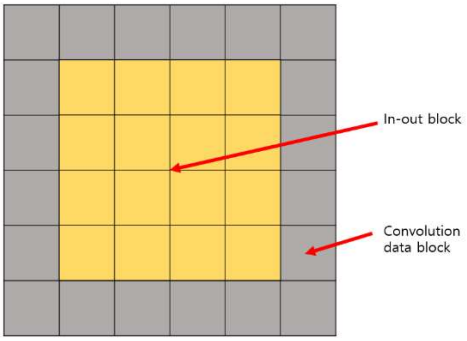

Figure 3_block convolution

I made Convolution_block function to solve Convolution_in's problem. In every Convlution_in function call, every filter value is called from *filter array and it takes a lot of time. Also, one pixel is called 9 times, when near pixel calculates convolution. So, in Convolution_block function, to calculate 4*4 block at once, stores all 6*6-pixel values and filters. Then, there are less variable call repeated, so this method makes program faster. Then we can think that bigger block makes better performance, But 8*8 Convolution_block is slower then 4*4 Convolution_block. Because 8*8 convolution_block should store too many data, so misses occur which makes program very slow.

2. Optimization rate

I used sharpening filter 20 times and used average as optimization rate.

rate/method	Convolution_in, Unroll=8	4*4 block convolution	8*8 block convolution
Img_128	3.76	4.25	4.133
Img_256	3.415	4.2	4.132
Img_512	3.95	4.8	4.6
Img_768	3.79	4.6	4.5355
Img_1024	3.42	4.55	4.43

As I said before, 4*4 block convolution filter showed best performance. Block calculation makes program faster, but if Block size is too big, program may be slower then not using Block method. Interesting thing is that performance is maximized when image size is 512*512. I think this is because I separated basic convolution function to 3 functions. When image is too small, then we call too many convolutions(basic) or convolution_in instead of convolution_in or convolution_block, so optimization rate close to optimization rate of convolution or convolution_in. And if size is too big, I think basic convolution filter's performance is increased. Because of that our convolution_in or convolution_block function's performance is fixed, my functions seemed to be relatively slow.

3. How to make program faster

I wanted to use SIMD to make best performance, but I could not because there are lot of time spend to load data from struct, and store data to struct. So, I could not optimize SIMD code and SIMD code was slower than my basic code. I got optimization rate 3.2, as 8*8 block convolution, Img_128. This is very slower than my 8*8 block convolution code. If I can load and store data with SIMD well, this program can be very faster.

And this program can be faster if filter conditions are limited.

First, if filter is Symmetric (many filters in computer vision is Symmetric, so we can use correlation instead of convolution), we do not have to load all filter value. We should load 3 values. edge, side, center. This makes filter load 3 times faster. Second, if filter is limited below and upwards, we can use fixed point rather then float. Float calculation is slower than fixed point, so this can make calculation time very quick. Third is very strict condition. Like gaussian and identity, average filter, every filter value is positive, and their sum is 1. Then, we do not have to check if their value exceeds 0-255. Since we must check this to all point, this can make program very quick.