






Implementation Results

After implementing several optimizations, I have collected optimization result data using the following commands sequentially.

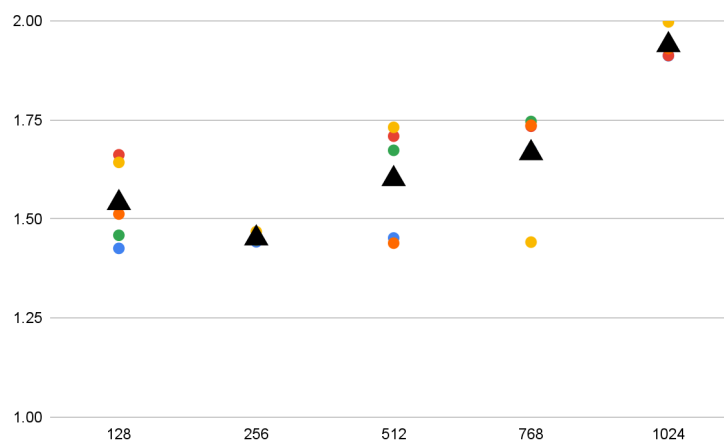
```
# run.sh

make clean
make all
./bmpfilter img_128.bmp ./img_output_128.bmp
./bmpfilter img_256.bmp ./img_output_256.bmp
./bmpfilter img_512.bmp ./img_output_512.bmp
./bmpfilter img_768.bmp ./img_output_768.bmp
./bmpfilter img_1024.bmp ./img_output_1024.bmp
```

Overall, the optimized implementation has shown up to 2.017 times speedup, and the results have varied when running multiple times. Following are the results obtained from running the shell commands above 5 times.

Run	img_128.bmp	img_256.bmp	img_512.bmp	img_768.bmp	img_1024.bmp
1	1.426139	1.442672	1.452302	1.673972	1.912317
2	1.662265	1.446921	1.709288	1.734344	1.912953
3	1.643172	1.468295	1.731625	1.441882	1.997676
4	1.459171	1.447112	1.673571	1.746372	1.943103
5	1.512746	1.452935	1.439309	1.736842	1.930391
Average	1.540699	1.451587	1.601219	1.666682	1.939288
Output					

The result shows an overall trend of increasing speedup rate as image size changes from 256 to 1024. The trend might have been made from the cache size and locality problem of the storage structure, and using the single-line row-wise tiling method (which will be explained in Implemented Optimizations - 2) might have been helpful as the size of the image increases.



The graph of 5 trials, with the average value drawn as a triangle point for each image size.

Optimization Approach

Implemented Optimizations

1. Removal of Dynamic Memory Allocation -> about 5~15% speedup

The original `Convolution` function in `filter_baseline` dynamically allocates memory for each pixel result using `malloc` and frees it with `free`. However, this dynamic allocation might cause overhead - so I used a `tmp_output` array to store intermediate results for each row of the image, avoiding the need for per-pixel memory allocation.

2. Loop Reordering, Pre-computation with Row-wise Tiling Method -> about 30% speedup

Row-wise tiling is an optimization technique that can be used when you process image row by row instead of pixel by pixel. The original filter code have been calculating convolution pixel by pixel, but as I reordered loops for better locality, I was able to use row-wise tiling method. This helps to use higher locality calculation.

3. Reduce Branching Access -> showed about 20% speedup

I removed every IF statement and instead used logical operands and `fmax/fmin` function to reduce branching access failures, which might lead to flush - a huge memory overhead.

4. Reduce Function Calls -> didn't calculate speedup by itself.

The optimized convolution is a single function, so no extra memory or jump is needed for the function calling every iteration.

5. Temporary Buffer-based Locality Improvement -> showed about 20% speedup

In the original code, each pixel is processed independently, and the final RGB values are immediately clamped and stored in the output array. I changed it to using a temporary buffer `tmp_output`, which stores the intermediate convolution results for each channel (R, G, B) in a row. This allows for more efficient memory access and simplifies the clamping step to a single operation per channel at the end of each row's processing.