Jason Drain
CSCE 569
Assignment 1

Implementation:

For mm, there were 4 different cases leading to 4 different equations that governed my implementation:

- When $A_{\text{row major}}$ and $B_{\text{col major}}$ $C[i*M+j] = \Sigma_k A[i*K+k] * B[j*K+k]$
- When $A_{\text{col major}}$ and $B_{\text{row major}}$ $C[i*M+j] = \Sigma_k A[i+k*N] * B[j*K+k]$
- When $A_{\text{col major}}$ and $B_{\text{col major}}$ $C[i*M+j] = \Sigma_k A[i+k*N] * B[j*K+i]$
- When $A_{\text{row major}}$ and $B_{\text{row major}}$ $C[i*M+j] = \Sigma_k A[i*K+k] * B[k*M+j]$

In this function, i, and j are indices for the rows of A and columns of B respectively, while k is an index over the length of the rows and columns. K is the length of the rows and columns, N is the number of rows of A, and M is the number of columns of B.

For mv, there were only two cases:

- When $A_{\text{row major}}$ $C[i] = \Sigma_k A[i*M+j] * B[j]$
- When $A_{\text{col major}}$ $C[i] = \Sigma_k A[i+j*N] * B[j]$

In this function, i is an index for the rows of A and j is an index over the length of rows of A and the length of B. N is the number of rows in B.

For both mm and mv I used these basic equations to define my function based on how the arrays were laid out in memory. I used three for loops to iterate over the data, sum it up, and store it in the result array.

For the histogram function, I first split the image into its three BGR planes. I then iterated over the underlying data in the Mat object that OpenCV uses to store images. Then, based on intensity, I incremented the histogram array at the index corresponding to the intensity. This was the most straightforward function, in my opinion, and the most challenging part was learning how to use the associated OpenCV functions and objects.

Finally, for the smoothing function, I iterated over all pixels except for those at the edges (I only implemented the function for 3x3 filters; I would need to exclude more if I went up to larger ones) and got a sum by applying the filter to each pixel around the one in question. I then got the average of this sum and set the pixel equal to that average.

Figures:

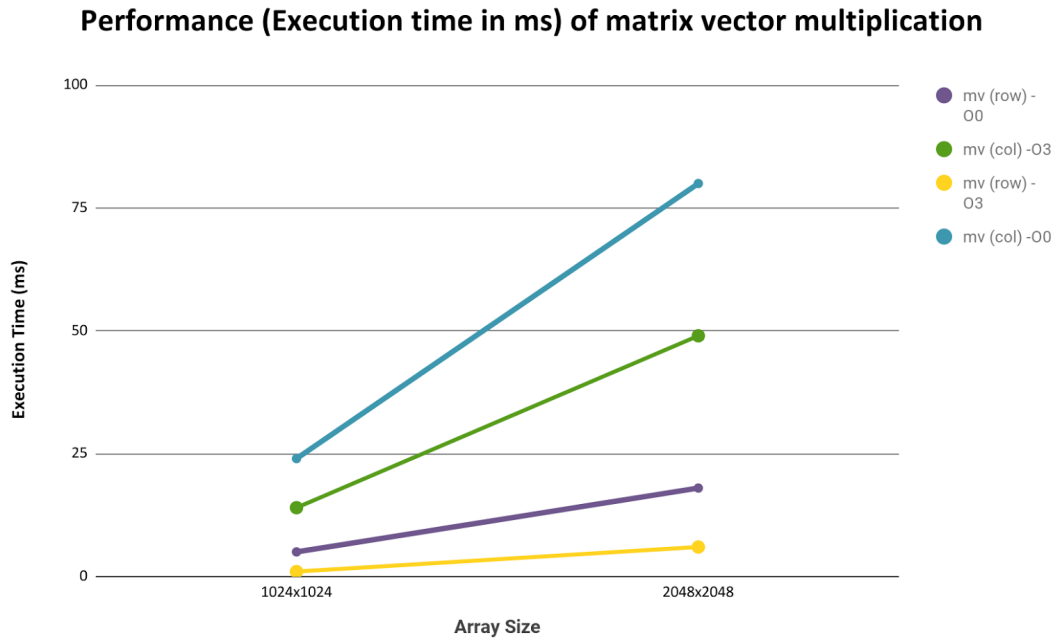

Fig. 1 -- Execution time of mv for square matrices of size 1024 and 2048 with level 0 and 3 compilation optimization. The matrices also vary in whether they are stored in memory as column or row major.

Fig. 2 -- Execution time of mm for square matrices of size 1024 and 2048 with level 0 and 3 compilation optimization. The matrices also vary in whether both are row major or both are column major, and whether the first is row major or the second is row major.

CPU and System Info:

My CPU is an Intel i3-2350M. It runs at 2.30 GHz, has 3 MBs of cache, and uses 4 floating point instructions per cycle. It has two cores, but since this program is not parallel, that is not relevant. I have 6 GBs of Memory. I am running Ubuntu 16.04 LTS.

Peak Performance:

$CPU_{Peak} = CPU_{speed} * IPC$
$CPU_{Peak} = 2.30\ GHz * 4\ Instructions = 9.20\ GFlops$

Efficiencies[1]:

For these functions, the most efficient was mv, with an efficiency of 18.99%. The next most efficient was mm, at 8.26%. The Histogram function came in at 2.06% and the least efficient function was Smoothing at .213%.

In the case of mv, the (relatively) high efficiency probably stems from the fact that data is accessed in the rows in a sequential manner, allowing for caching, and then the vector being multiplied is accessed sequentially too, as well as only needing to be accessed once in the beginning, and being cached for the remainder of the program.

Similar logic applies to mm, but there are multiple columns in the second matrix, so more memory access needs to occur. For the Histogram function, memory is accessed sequentially in the base image, but the actual histogram array is accessed in a non sequential manner. Finally, for the smoothing function, memory access is pretty non sequential, as each point requires access to the 9 adjacent points around it.

As for the combination of row major and column major, mm was most efficient when A was row major and B was column major. This is because A is having its rows accessed, and B is having its columns accessed. Therefore, we want these to be sequential in memory so that the CPU can cache the data. Along with this principle, we see that the least efficient is A as column major and B as row major. In mv, the layout of B is already sequential as it is one dimensional, but as expected, having A in memory as row major produces the fastest execution time.

Another optimization was to compile with the flags -O0 and -O3.[2] The -O0 flag tries to reduce code size and execution time, but does not want to drastically increase compilation time. The -O3 flag performs these same optimizations, as well as others that can significantly increase compilation time, and maybe even involve space for speed tradeoffs. However, the data show that compiling at the 3rd optimization level increases performance in a non trivial way.

---

[1] Reported efficiencies for mv and mm are for programs compiled with the -O3 flag. For mv, this efficiency is with a row major matrix, and for mm it is A row major, B column major.
[2] Information on the flags was gathered here: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html