LOCALITY

Authors: William Magann (wmagan01) and Javier Laveaga (jlavea01)
Date: Sunday February 18, 2024

Implementation Plan

ppm trans (performing simple image transformations)

Main

- Check to see which option is provided
- If no option is selected, call the function responsible for 0 degree rotation.
- If file cannot be opened, stderr and exit with exit_failure
- Testing: We will provide user commands that include several different combinations of the options available to the user, such as including or excluding a major or filename.

Reading in the file:

- We will use Pnm_ppmread to read in the file.
    - nm_ppm Pnm_ppmread(FILE *fp, A2Methods_T methods);
- If no specified argument is given, the file will be read from stdin.
- Row major and column major will mean that the methods suite of uarray2_methods_plain will be used.
- Block major will mean that the methods suite of the uarray2_methods_blocked will be used.
- Testing: We will test the empty file and specify plain or blocked as arguments for the ppmread. We will also test by inputting a file through stdin.

Rotations:

- For every rotation, depending on the specified argument (row major, column major, block major), we will call the appropriate functions indirectly through the function pointers in the methods suite.
- Each rotation function will take in a boolean for whether to start a timer or not depending on user input. For following functions, timer functionality is written as the rotation will go with the timer included. If statements will be used for when no timer should be used
- Testing: We will test this by including or not the timer feature as well as different majors.

Rotate 0 degrees

- Start Timer
- Output to timing file
- Return Array 2
- Testing
    - Use diff test on outputted file and original file to confirm that they are the same

Rotate 90 degrees

- Create new 2D array of same size
- Start timer

- Create apply function for mapping function that takes in a location and translates that pixel to (height − col − 1, row) in the new array which corresponds to a 90 degree rotations
- Row Major Order
    - Call apply function specified earlier
- Column Major Order
    - Call apply function
- Block major
    - Call apply function
- Stop timer and put output to timing-file
- Free old 2D array
- Return new 2D array
- Testing
    - Create a small ppm file that we can manually rotate and compare the 90 degree rotation results to our manually rotated image.
    - If that works we can test on larger files and display the results to see if they appear to be correctly rotated 90º.
    - Do this process on row, column, and block major mapping to make sure all three work

Rotate 180 degrees

- Create new 2D array of same size
- Start new timer
- Create apply function for mapping function that takes in a location and translates that pixel to ((width − col − 1), (height − row − 1)) in the new array which corresponds to a 90 degree rotations
- Row Major Order
    - Call apply function specified earlier
- Column Major Order
    - Call apply function
- Block major
    - Call apply function
- Free old 2D array
- Return new 2D array
- Testing:
    - Similar to the 90 degree rotation, start by rotating a small ppm file 180 degrees that we can manually compare the output rotation to to check if the rotation operated successfully.
    - Then move onto larger files and use the display function to see if they appear correctly rotated 180 degrees.

    - Do this process on row, column, and block major mapping to make sure all three work

Timer:

- Stop timer and put output to timing-file (create file using fopen)
- Output the number of nanoseconds taken, and compute the time per pixel.
- width and height of image, and amount of rotation.
- FILE *timings_file = fopen("timing_file", "a")
- Write to file named in -timing <timing_file>
- Optional: width and height of image, and amount of rotation.
- Testing
    - After we have the rotation functions working, we will then test the timer function
    - First we will create a simple function unrelated to rotations like one that computes the sum of two numbers to see if we can correctly output times to a file.
    - Once that step is completed we will use the timer function for each row, column, and block major for each of the rotations described above and see if the output is correctly formatted in output.

After each Rotation

- Write the final transformed image to standard output (in binary format).
- Testing: We will create our own large images using JPEG files with djpeg and pnmscale and get a desirable size. One that gets enough CPU time but is not too big.
    - We will display the images to see the before and after results after rotations.

Part D: Analyze Locality and Predict Performance

	row-major access (UArray2)	column-major access(UArray2)	blocked access (UArray2b)
90-degree rotation	4	6	2
180-degree rotation	3	4	1

- In UArray2, the structure of the 2D array is a 1D UArray holding more 1D arrays of every row in the 2D array. Because of this fact, the elements in each row are stored next to each other in memory while each row is farther from other rows in memory. Thus row major has more cache hits than the column major because while mapping by row major, you are iterating through elements that are closer together in memory. This means that when blocks are moved to the cache, those elements will all be there together, increasing the number of hits.
- On the other hand, column major will be accessing elements that are not close by in memory because of the way UArray 2 is implemented. Going from one element to the other means jumping from one UArray in memory to another, and thus you might not have all those blocks already in the cache. The cache will have to read from memory more times to get access to the next element in a different Uarray. Since we are assuming that the images are too large to fit in the cache, this often will mean that memory will have to be evicted for a whole new block of memory to be placed into the cache.
- For direct mapped caches, if two rows of the array lie in different blocks of memory and they are mapped to the same line in the cache, it will mean that the cache will need to write an evict for every element in that row (when doing a column major)
- Knowing this, we believe that of the 4 UArray2 rotations, the row-major access 180-degree rotation will have the highest percentage of cache hits. This is because reading through the file in row major order is faster than column major order. However, we believe it will have more cache hits than the row major 90-degree rotation as for the 180-degree rotation, when you are mapping into the new image array, you will be writing the new lines in row major order as well as rows all have the same elements and are simply moved and reversed. This differs in the 90 degree rotation where because rows and columns are swapped, you will need to write the lines in the new image in column major order resulting in more cache misses.
- As the 90-degree row-major rotation has a row major read and then column major write, we believe it will be roughly the same amount of time as the column major read of the 180-degree rotation. Like the 90-degree row major rotation, the 180 column major

rotation involves one slower column major read and one row major write as the 180 rotation is written in row major order. Thus both rotations involve one row major iteration and one column major iteration and will be about the same percentage of cache hits.

- We believe the 90 degree column major rotation to have the worst cache hit rate because it requires both a column major read and column major write. For reasons explained above, column major has less cache hits than row major, and thus doing a column major iteration twice will be the slowest way to rotate an image.
- We believe that both UArray2b rotations will mostly be faster than all the UArray2 rotations. For this conclusion, we assumed that the block size would be very large approaching the maximum block size for reasons we will explain later. The way out blocks are structured, each block is represented by a 1D UArray where each element is next to each other in row order with the last column of one row being adjacent in the array to the first column element of the row below it. Thus iterating through each block will have a high cache hit rate as all the elements are next to each other. With a large block size the number of times we will need to make the larger jump from one block to the next will be most likely smaller than the number of jumps we would need to make between rows, hence why we believe it would have more cache than row major order.
- The reason we believe the 180 blocked rotation will have more cache hits than the 90 degree one is that within each block, the 180 rotation writes to the new block in row major order while in the 90 degree rotation it writes to the new block in column major order. This is the same reasoning as we used for the UArray2 mapping.
- For this we assumed the blocksize to be large as the image is explicitly said to be very large, and without a specified block size determining the relative cache hit rate of the block mapping would be impossible. For example if the blocksize was 1, then the block mapping would be as slow as column major. However, as the default block size is as large as possible we felt that assessing the cache hit rate with this large block size to be the most appropriate.