**Implementation Plan**

1. Download a small initial test ppm image. Name it test_small.ppm. Download a large test ppm image. Name it test_large.ppm.
2. Reading & writing input from file
    a. Include pnm.h for reading and writing ppm files
    b. Use Pnm_ppmread to read in a small ppm image. Store the returned Pnm_ppm struct in a variable called ppm_data. Print out header properties of the image and the actual 2D array from ppm_data.
    c. Test:
        i. Input: a test ppm image with magic number P6, width and height 400, and denominator 255
        ii. Expected output: P6 400 400 255
    d. Call Pnm_ppmwrite and pass in ppm_data to write out the ppm image
    e. Test:
        i. Input: test_small.ppm
        ii. Expected output: the same ppm image
        iii. diff output ppm with original ppm
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm
    f. Note: The existing code in ppmtrans already filters invalid input so these cases are already handled for us
3. 0-degree image rotation:
    a. If the input rotation is 0 degrees, do not alter the ppm. Instead, immediately call Pnm_ppmwrite as we did in Step 2.
    b. Test:
        i. Input: a test ppm image with -rotate 0 specified
        ii. Expected output: an identical ppm image
        iii. diff output with input to ensure the ppm does not change
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm
4. 90 degree rotation:
    a. Check whether the original A2 (we will use this as shorthand for A2Methods_UArray2 from now on) stored in pnm_data is plain or blocked by checking its blocksize.
    b. Make a new A2 instance transformed_array.
    c. If original A2 is plain:
        i. Allocate transformed_array with methods->new to have same size as original array, but new width = original height and new height = original width
        ii. Call map_row_major function on original A2
    d. If original A2 is blocked:
        i. Allocate transformed_array with methods->new_with_blocksize to have same size and blocksize as original array, but new width = original height and new height = original width

ii. Call map_block_major function on original A2

e. For both map functions, apply function will be the same:
   i. Closure will be a pointer to the new A2 instance, transformed_array.
   ii. Use math from Geometric Calculations (Section 2.6 of spec) to set indices of transformed_array to values of original A2

f. Test with a plain A2:
   i. Input: ./ppmtrans test_small.ppm -rotate 90 -row-major
   ii. Expected output: an identical image rotated 90 degrees clockwise
   iii. diff output with input to ensure the ppm does not change
   iv. Use valgrind to ensure no memory leaks
   v. Repeat previous steps with test_large.ppm

g. Test with a blocked A2:
   i. Input: ./ppmtrans test_small.ppm -rotate 90 -block-major
   ii. Expected output: an identical image rotated 90 degrees clockwise
   iii. diff output with input to ensure the ppm does not change
   iv. Use valgrind to ensure no memory leaks
   v. Repeat previous steps with test_large.ppm

5. 180 degree rotation
   a. Implementation is similar with 180 degree rotation, except when allocating the new A2 instance transformed_array, the width and height remain the same as in the original A2. Also, in the apply function the math will be different.
   b. Test with a plain A2:
      i. Input: ./ppmtrans test_small.ppm -rotate 180 -row-major
      ii. Expected output: an identical image rotated 180 degrees clockwise
      iii. diff output with input to ensure the ppm does not change
      iv. Use valgrind to ensure no memory leaks
      v. Repeat previous steps with test_large.ppm
   c. Test with a blocked A2:
      i. Input: ./ppmtrans test_small.ppm -rotate 180 -block-major
      ii. Expected output: an identical image rotated 180 degrees clockwise
      iii. diff output with input to ensure the ppm does not change
      iv. Use valgrind to ensure no memory leaks
      v. Repeat previous steps with test_large.ppm
6. Repeat all rotation tests with the large test ppm files in /comp/40/bin/images/large. Expected outputs will follow the same patterns as above. Repeat each test with valgrind to ensure no memory is leaked.
7. -time
   a. Follow instructions in (Locality): Timing Instructions.
   b. Test:
      i. Input: ./ppmtrans mobo.ppm -rotate -time 180 -block-major (where mobo.ppm is a large ppm from /comp/40/bin/images/large)
      ii. Expected output: a file created that contains the time for actual rotation
      iii. Compare with different tests with different rotation/image size to make sure the time makes sense.

iv. Use valgrind to ensure no memory leaks

**Locality Analysis & Performance Predictions**

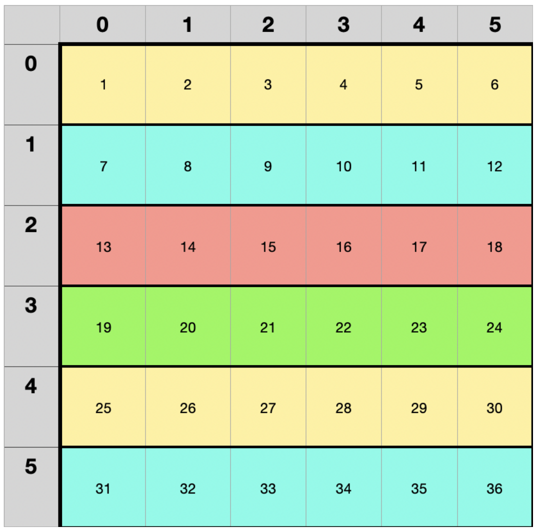
Assumptions

- Comparing the cache hit/miss rate *for reads only*
- One-dimensional cache with cache block of size 32 bytes (enough to hold 8 ints)
- Block size: 4
- Images being rotated are 100 KB

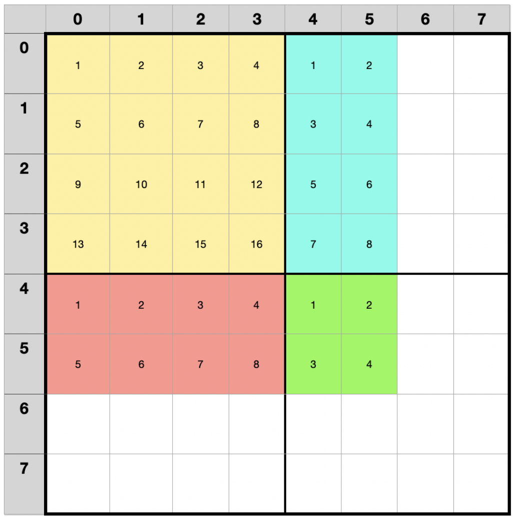
| | **Row-major access (UArray2)** | **Column-major access (UArray2)** | **Blocked access (UArray2b)** |
|---|---|---|---|
| **90-degree rotation** | 1 | 6 | 3 |
| **180-degree rotation** | 1 | 6 | 3 |

Explanation of rankings

- When reading data, the cache block will store the memory of one element and the memory of elements in adjacent memory locations
- Since we are only measuring the cache hit rate for reading and not writing, in each column above the hit rate will be consistent, i.e. it will be the same regardless of what rotation is performed.
- The solution UArray2 implementation stores data in a row-major order, i.e. it is made up of a vertical UArray where each element is a horizontal UArray.
    - This means that row-major access will have a higher hit rate than column-major access since the elements in each row of the UArray2 implementation are stored adjacent to each other in memory – they have good spatial locality. For example, when accessing the element at (0,0) in the simplified diagram below, its nearby elements will already be stored in the cache, meaning finding them will be easier when we need to access them, so the cache hit rate will be higher.
    - Since the UArray2 implementation stores the elements of each row adjacent to each other in memory, accessing the elements in column-major order means that the benefits of spatial locality will not apply – the elements along each column are *not* adjacent to each other in memory (for example (0,0) and (0,1) are 5 elements apart in the diagram below). To access the elements of the UArray2 implementation in column-major order, for every new column, a new row has to be moved from main memory to the cache. As a result, the cache hit rate will be low and the cache miss rate will be high.



- UArray2b stores data in blocks. Reading the elements in a UArray2b means accessing the elements block-by-block.
    - For blocked access, whether or not there are spatial locality benefits depends on the size of the blocks and the specific implementation. For the first line in the simplified block diagram below, since elements (0,0) to (3,0) are stored in the same block, we know they are stored near each other in memory and have good spatial locality, meaning we will have a higher rate of cache hits than with column-major access. However, once we reach element (4,0), we will need to bring the next block into the cache from main memory. Since there are no guarantees that the next block will be adjacent to the previous block in memory, we do not have the benefit of spatial locality between blocks. Reading a complete row requires accessing several different blocks, so we know the spatial locality will be worse than with row-major access in a UArray2 with the same width and height. Therefore, the cache hit/miss ratio of UArray2b lies somewhere between row-major access in a UArray2 and column-major access in a UArray2.


0 1 2 3 4 5 6 7
0 1 2 3 4 1 2
1 5 6 7 8 3 4
2 9 10 11 12 5 6
3 13 14 15 16 7 8
4 1 2 3 4 1 2
5 5 6 7 8 3 4
6
7