locality

Problem statement

Implement blocked two-dimensional arrays, which will be used to evaluate the performance of image rotation using three different array-access patterns with different locality properties.

Use Cases

This program will be used by clients who desire to perform rotational and transformational operations on images.

Assumptions and Constraints

This program assumes that a supplied file name will be a valid `pnm` file. The program will print the transformed image to stdout.

Implementation Plan for `ppmtrans.c`

* Throughout this plan, we omit repeated mentions of checking for `NULL` arguments or invalid input files (e.g., file not found). In practice, we will consistently verify that all memory allocations succeed and handle any unexpected `NULL` pointers or errors by terminating the program appropriately.

1. Program Setup
   - Ensure that `ppmtrans.c` includes the necessary headers: C libraries, Hanson's `assert.h`, and the course-provided `a2methods.h`, `a2plain.h`, `a2blocked.h`, and `pnm.h`.
   - Verify that the Makefile compiles `ppmtrans.c` into an executable.
     (a) test input: `make ppmtrans`
     Test output: executable `ppmtrans` built with no errors.
2. Main Function and Create A2 Instances.
   - Open the input file if provided or otherwise default to `stdin`.
     – If the file fails to open, print an error message to `stderr` and exit with `EXIT_FAILURE`.
   - Call `Pnm_ppmread(FILE *fp, A2Methods_T methods)` to read the image into memory. Use the chosen `methods` (this could be plain or blocked).
   - If plain methods are selected: `methods->new` creates a `UArray2` with no `blocksize`. (`blocksize` = 1 by invariant).
   - If blocked methods are selected: `methods->new` calls `UArray2b_new`, which will set the `blocksize` using our `uarray2b.c` implementation.
     (a) Test input: `./ppmtrans -row-major -rotate 0 example1.ppm`
     Test output: Creates a new, correct A2 instance with methods correctly set for dealing with a UArray2_T under the hood.
     (b) test input: `./ppmtrans -rotate 0 example.ppm`
     Test output: We will verify that width and height match the dimensions of `example.ppm` by printing the width and height values. We will test that by default the A2 is a UArray2.
     (c) Test input: `./ppmtrans -block-major -rotate 0 example.ppm`
     Test output: We will verify that the A2 is a UArray2B_T
3. Copy the given image to A2
   - While reading the ppm file, use `methods->at()` to access every element of the A2 and insert using the values in the `ppm` pixel array. Confirm each slot in the array holds a valid `Pnm_rbg` value.

- Validate that each pixel has been placed in the expected index in the A2 data structure.
- Ensure the new A2 instance has been created with the same width and height as the original image.
- With a valid A2, the `rotate` function is ready to be called (implementation steps detailed below).

(a) Test input: Pass a file with multiple different pixel values (intensities)
Test output: The A2 should contain the same pixels in the same positions with the same values.

For all of the following rotations, each function must use the `map` (chosen mapping function) associated with the data structure the A2 is built from rather than nested for-loops, as per the spec.

4. Implement 0-degree rotation.
   - Function signature: `void rotate_0(Pnm_ppm *image)`
   - The output should match the original `ppm`.
   - Exit: `EXIT_SUCCESS`.

   (a) test input: `./ppmtrans -rotate 0 example2.ppm`
   Test output: Use `diff` to ensure that the output is identical to the input file.

5. Implement 90-degree rotation
   - Function signature: `Pnm_ppm rotate_90(Pnm_ppm image_src, A2Methods_mapfun *map)`
   - Allocate a new A2 with swapped dimensions (`width = image_src.height` and `height = image_src.width`) use the selected `map` function to visit each pixel.
   - Map each pixel from the image stored in the A2 (col, row) to $(new\_col, new\_row) = (height - 1 - row, col)$.
   - Update the `Pnm_ppm` struct to point to the new array and free the old one.

   (a) test input: Pass a 1x2 `ppm` that has two rows and one column: $\begin{bmatrix}1\\2\end{bmatrix}$
   Test output: A `ppm` that has one row and two columns: $\begin{bmatrix}2 & 1\end{bmatrix}$

   (b) test input: Various sizes and dimensions of `ppm`'s.
   Test output: Ensure that the dimensions are reversed and the data has undergone the appropriate transformation.

6. Implement 180-degree rotation
   - Function signature: `Pnm_ppm rotate_180(Pnm_ppm image_src, A2Methods_mapfun *map)`
   - Allocate a new A2 with the same dimensions as the original image using the selected `map` function to visit each pixel.
   - `Rotate_180` will map each pixel from the image stored in the A2 (col, row) to $(new\_col, new\_row) = (width - 1 - col, height - 1 - row)$. All pixels will be flipped horizontally and vertically.
   - Update the `Pnm_ppm` struct to point to the new array and free the old one.

   (a) test input: Pass a 1x2 `ppm` that has two rows and one column: $\begin{bmatrix}1\\2\end{bmatrix}$
   Test output: A `ppm` that has one row and two columns: $\begin{bmatrix}2\\1\end{bmatrix}$

   (b) test input: Various sizes and dimensions of `ppm`'s.
   Test output: Ensure that the dimensions are reversed and the data has undergone the appropriate transformation.

7. implement 270-degree rotation
   - Function signature: `Pnm_ppm rotate_270(Pnm_ppm image_src, A2Methods_mapfun *map)`
   - Allocate a new A2 with the same dimensions as the original image using the selected `map` function to visit each pixel.

- `Rotate_270` will map each pixel from the image stored in the A2 (col, row) to $(new\_col, new\_row) = (row, width - 1 - col)$.
- Update the `Pnm_ppm` struct to point to the new array and free the old one.

(a) test input: Pass a 1x2 `ppm` that has two rows and one column: $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$
Test output: A `ppm` that has one row and two columns: $\begin{bmatrix} 1 & 2 \end{bmatrix}$

(b) test input: Various sizes and dimensions of `ppm`'s.
Test output: Ensure that the dimensions are reversed and the data has undergone the appropriate transformation.

8. Timing support.
   - If the `-time <file>`, tag was provided, then wrap the rotation call with the provided timing library functions so we can see the amount of time it took to run the rotation.
   - Sum the total CPU time and time per pixel and add it to the given file.
   - TEST: Run with and without
   - Exit: `EXIT_SUCCESS`.
     (a) Test input: `./ppmtrans -rotate 90 -time out.txt example3.ppm`
     Test output: A line of timing data that results from performing a rotation of 90 degrees.

9. Final output and cleanup.
   - All A2 accesses and mappings will be handled by the `methdos` suite.
   - Call `Pnm_ppmwrite(stdout, image)` to write the result in the expected binary format.
   - Free all allocated arrays using `methods->free()` and call `Pnm_ppmfree()`.
   - Close the input file if stdin was not used.
   - Exit: `EXIT_SUCCESS`.
     (a) Test input: `./ppmtrans -flip horizontal example5.ppm`
     Test output: error message on `stderr` and `EXIT_FAILURE` as no output file was provided.
     (b) Test input: `./ppmtrans -badcommand example5.ppm`
     Test output: error message on `stderr` and `EXIT_FAILURE` since a bad command was given.

Explanation of UArray2b Representation and Invariants

Our `UArray2b` will be built using the suggested architecture. As such, this data structure will be implemented using a single `Uarray2_T` that contains multiple `Uarray_T`'s. Each `Uarray2_T` represents a block and every `Uarray_T` represents the data contained within a block. We will use William Goldman and Andrew Bacigalupi's implementation for `Uarray2_T`.

The invariants in this document will refer to the variables below which will be defined following a call to `UArray2b_new()` or `UArray2b_new_64K_block()`.

1. `width` refers to the number of columns in the x dimension of the `Uarray2b`.
2. `height` refers to the number of rows in the y dimension of the `Uarray2b`.
3. `blocksize` refers to the cells per side in a full block, where blocksize ≥ 1.
3. `elem_size` refers to the size in bytes of each element > 0.
4. `blocks` represents a `Uarray2` of `Uarray_T`'s with dimensions: `block_w` by `block_h`.

In our representation, we consider each `UArray2_T blocks` as a block in the blocked 2-dimentional array we are building. Within each block is a `UArray_T` that stores the block's elements in contiguous memory.

When describing the invariants contained in our program, we will reference values that define various aspects of our data structures. Those quantities are described below:

Derived quantities

$$\text{blocks\_w} = \left(\frac{\texttt{width}}{\texttt{blocksize}}\right) \qquad \text{blocks\_h} = \left(\frac{\texttt{height}}{\texttt{blocksize}}\right)$$

For a block with block-index (b_x, b_y) (where $0 \leq b_x <$ blocks_w and $0 \leq b_y <$ blocks_h) we define

$$\text{local\_w}(b_x) = \min(\texttt{blocksize}, \texttt{width} - b_x * \texttt{blocksize}) \qquad \text{local\_h}(b_y) = \min(\texttt{blocksize}, \texttt{height} - b_y * \texttt{blocksize}).$$

Note: min is used to indicate that the smaller of the two values in the parentheses will be chosen (either the data fills the block, or the block is partially full). These give the actual number of valid columns/rows held by that block (edge blocks may be smaller than `blocksize`).

Representation invariants

1. **Parameter bounds.** $\texttt{width} \geq 0$, $\texttt{height} \geq 0$, $\texttt{blocksize} \geq 1$, $\text{elem\_size} > 0$. (In particular, creating a UArray2b with $\texttt{blocksize} < 1$ is a checked run-time error per the spec.)
2. **Blocks container shape.** `blocks` is non-NULL and its dimensions satisfy blocks_w $*$ blocks_h $=$ the number of blocks allocated. In other words, the outer `UArray2_T` that stores block objects has width blocks_w and height blocks_h.
3. **Per-block sizes match client dimensions.** For every block at index (b_x, b_y):
   - the block's `UArray_T` length equals $\text{local\_w}(b_x) * \text{local\_h}(b_y)$,
   - for non-edge blocks ($b_x <$ blocks_w $- 1$ and $b_y <$ blocks_h $- 1$) we have local_w $=$ local_h $=$ `blocksize`.

   This ensures there is no unused storage counted as part of the logical array (the mapping functions must not visit padding/extra cells).
4. **Element size uniformity.** Every block's elements use the same `elem_size` specified at creation. To do this we ensure each `UArray_T` for a block was created with size `elem_size`.
5. **Contiguity within a block.** The storage for a block's elements is a single contiguous allocation of local_w $*$ local_h $*$ `elem_size` bytes; indexing within the block is row-major (row offset * local_w + column). This guarantees that, for a valid within-block coordinate (i, j) with $0 \leq i <$ local_w, $0 \leq j <$ local_h, the address returned by `UArray2b_at(x,y)` points into that contiguous block buffer.
6. **Index mapping correctness.** For any client coordinates $0 \leq x < \texttt{width}$ and $0 \leq y < \texttt{height}$,

   $$b_x = \left(\frac{\text{x}}{\texttt{blocksize}}\right) \qquad b_y = \left(\frac{\text{y}}{\texttt{blocksize}}\right)$$

   and the in-block coordinates are

   $$i = x \ \% \ \texttt{blocksize}, \quad j = y \ \% \ \texttt{blocksize}.$$

   The element returned by `at(x,y)` is the element at block (b_x, b_y), local row j, local column i. For edge blocks, the invariants above ensure $i < \text{local\_w}(b_x)$ and $j < \text{local\_h}(b_y)$.

7. **Block-major mapping order** The `map_block_major` implementation visits blocks in block-major order; for each block it visits the block's valid cells in row-major order. It must not visit padding cells that lie outside the region defined by `width` $*$ `height`.

8. **Allocation** The UArray2b object "owns" all per-block allocations: every block's internal buffer and the outer `blocks` container are allocated on `UArray2b_new` and must be released by `UArray2b_free`. After `UArray2b_free` returns, no pointers into the former object remain valid.

Answers to Part D

Assumptions

- Assume integers (each pixel in the image) are 4 bytes in size.
- Assume the cache can hold up to a single row's worth of data before evicting (the cache is 1 line). In our example, the cache size is 16 bytes (4 bytes per pixel * 4 pixels per row).
- As implied above, assume that the images being rotated are much too large to fit in the cache. In our example, the image is 4x4 pixels.
- Assume the image has been properly built using either a `UArray2` or a `Uarray2b`.
- Given the above assumption, assume data is stored contiguously in memory for both the `UArray2` and the `Uarray2b` implementation.

The full analysis of this problem can be summarized in the following thought:

Because we assume that data is stored contiguously in memory for each implementation (`UArray2` or `UArray2b`), any read that follows the same order as the implementation will have ideal performance. In this way, a row-major read for a `UArray2` will have optimal performance as will a block-major read for a `UArray2b`. A col-major read over a `UArray2` will not access contiguous data, resulting in the worst performance of the 6 possible reads. **Data will be read in the exact same way for both 90 degree and 180 degree rotations.**

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

Table 1: Estimated cache hit rate for reads under different array access patterns.

In the table above, 1 is the best hit rate and 2 is the worst hit rate. To emphasize this idea we provide an example below:

Example

We assume the cache is 16 bytes in capacity. Below we analyze different access patterns for the images depicted in Figure 1 and Figure 2.

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

Figure 1: An image built using a `UArray2`

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

Figure 2: An image built using a `UArray2b`, `blocksize` of 2

Row-major Access of `UArray2`

1. Attempt to read 0. **Miss.** Pull in 16 bytes: {0, 1, 2, 3}.
2. Attempt to read 1. **Hit.**
3. Attempt to read 2. **Hit.**
4. Attempt to read 3. **Hit.**
5. Attempt to read 4. **Miss.** Evict previous block. Pull in {4, 5, 6, 7}.
6. Attempt to read 5. **Hit.**
7. Attempt to read 6. **Hit.**
8. Attempt to read 7. **Hit.**
9. And so on for subsequent rows...

Every row that we read will be contiguous in memory, meaning the first element of each row that is read will be a miss for the cache, but every other index that is iterated over in the row will be a hit, since the cache can store the entire row. Therefore we will miss once per row.

Column-major Access of `UArray2`

1. Attempt to read 0. **Miss.** Pull in {0, 4, 8, 12}.
2. Attempt to read 1. **Miss.** Evict previous block. Pull in {1, 5, 9, 13}.
3. Attempt to read 2. **Miss.** Evict previous block. Pull in {2, 6, 10, 14}.
4. Attempt to read 3. **Miss.** Evict previous block. Pull in {3, 7, 11, 15}.
5. And so on for subsequent columns...

This pattern leads to frequent cache misses due to strided access. Reading in column major will cause a miss on every index read, because the integer being accessed each time is too far away in memory to get hit by the cache (the cache takes in the full row's worth of indexes, but the next index is just out of reach).

Block-major Access of `UArray2b`

1. Attempt to read 0. **Miss.** Pull in {0, 1, 2, 3}.
2. Attempt to read 1. **Hit.**
3. Attempt to read 2. **Hit.**
4. Attempt to read 3. **Hit.**
5. Attempt to read 4. **Miss.** Evict previous block. Pull in {4, 5, 6, 7}.
6. Attempt to read 5. **Hit.**
7. Attempt to read 6. **Hit.**
8. Attempt to read 7. **Hit.**
9. And so on for subsequent blocks...

Every block that we read will be contiguous in memory, meaning the first element of each block that is read will be a miss for the cache, but every other index that is iterated over in the block will be a hit, since the cache can store an entire block's worth of memory. Therefore we will miss once per block.

Summary of Results

Access Pattern	**Hits**	**Misses**	**Evictions**
Row-major (UArray2)	12	4	3
Column-major (UArray2)	0	16	15
Block-major (UArray2b)	12	4	3

Table 2: Comparison of cache performance under different access patterns (cache size of 16 bytes).

As depicted in Table 2, the performance for row-major access and block-major access is the same, and the performance for col-major access is the worst of the three.