CSC 411 Assignment: Locality and the costs of loads and stores

Designs and experimental estimates due Friday, Oct 20th, at 11:59 PM. Full assignment due Friday, Oct 27th, at 11:59 PM.

READ THIS ENTIRE DOCUMENT FIRST

Overview and purpose

This assignment is all about the cache and locality. You'll implement an image-rotation program which uses your `Array2` from the last assignment, which you'll then use to evaluate the performance of image rotation using three different array-access patterns with different locality properties.

The assignment has two parallel tracks:

1. On the *design and building* track, you will implement image rotation, which may require some changes to your `Array2` implementation.
2. On the *experimental computer-science* track, you will predict the costs of image rotations, and later measure them. Your predictions will be based on knowledge of the cache as covered in Chapter 6 of Bryant and O'Halloran and as covered in class.

Problems

Part A (design/build): `ppmtrans`, a program with straightforward locality properties

You now have a representation of two-dimensional arrays: `Array2`, which supports **column-major** and **row-major** mapping with either **column-major** or **row-major** storage. Being clear about your storage model is essential.

Using `Array2` implement program `ppmtrans`, which is modelled on `jpegtran` and performs some simple image transformations. Program `ppmtrans` offers a subset of `jpegtran`'s functionality. The image-transformation options you may support are as follows:

```
--rotate 90
      Rotate image 90 degrees clockwise.
```

```
   --rotate 180
          Rotate image 180 degrees.

   --rotate 270
          Rotate image 270 degrees clockwise (or 90 ccw).

   --rotate 0
          Leave the image unchanged.

   --flip horizontal
          Mirror image horizontally (left-right).

   --flip vertical
          Mirror image vertically (top-bottom).

   --transpose
          Transpose image (across UL-to-LR axis).
```

You **must implement both 90-degree and 180-degree rotations.** Other options may be implemented for extra credit; if you choose not to implement them, **reject** the unimplemented options with a suitable error message written to `stderr` and a nonzero exit code.

Significant requirements:

Your program must also recognize and implement these options:

```
  --row-major
         Copy pixels from the source image using iter_row_major
         (with an Array2)
  --col-major
         Copy pixels from the source image using iter_col_major
         (with an Array2)
```

- You **must use the row_major and col_major iterators defined in your Array2,** not nested for loops over ranges of coordinates. - So this is ok: `for (col, row, pix) in src.iter_row_major() { ... }` - This is not: `for col in 0..width { for row in 0..height { ... } }`
- For row-major and column-major mapping, you will use the `Array2` module, with whatever native storage (row- or column-major) you chose when you designed it.

Your `ppmtrans` should read a single `ppm` image either from standard input or from a file named on the command line. Your `ppmtrans` should write the transformed image to standard output. For help handling command-line options, see the suggested code at the end of this assignment.

Why this problem is interesting from a cache point of view:

If cells in a row are stored in adjacent memory locations, processing cells in a row has good spatial locality, but it's not clear about processing cells in a column. If cells in a column are stored in adjacent memory locations, processing cells in a column has good spatial locality, but it's not clear about processing cells in a row. In a 90-degree rotation, processing a row in the source image means processing a column in the destination image, and vice versa. Thus, the locality properties of 90-degree rotation are not immediately obvious. In a 180-degree rotation, rows map to rows and columns map to columns. Thus, whatever locality properties are enjoyed by the source-image processing are enjoyed equally by the destination-image processing. If you understand how your data structure works, then, you should find it easier to predict the locality of 180-degree rotation.

Part B (experimental): Analyze locality and predict performance

This part of the assignment is to be completed at the same time as your design work for parts A and B. Please **estimate the expected cache hit rate** for reads of each of the four operations in the table below. Assume that the images being rotated are **much too large to fit in the cache.**

	row-major access	column-major access
90-deg rotation		
180-deg rotation		

Your estimate should be a **rank between 1 and 4,** with 1 being the best hit rate and 4 being the worst hit rate. If you think two operations will have about the same hit rate, give them the same rank. For example, if you think that both column-major rotations will have the most cache misses and will have about the same number of cache misses, rank them both 3 and rank the other entries 1 to 2.

Justify your estimates on the grounds of expected cache misses and locality. Your justifications will form a **significant fraction** of your grade for this part.

Unfortunately, measuring hit rates is not so easy (although `valgrind` can do a lot). But what we are really interested in is the effect of locality on performance. We are therefore also asking you to **predict the relative performance** of each algorithm. But do make some simplifying assumptions:

- As before, assume that the images being rotated are **much too large to fit in the cache.**
- Assume that all function calls cost the same, and that each algorithm does the same number of function calls.
- Assume that the cost model for stores is approximately the same as the cost model for loads: if the store modifies a line already in the cache, the

cost is about one cycle, but if the store writes an address that is not already in the cache, it costs about as much as a cache miss. [This assumption oversimplifies the cache's behavior significantly, but it will be good enough to enable reasonable predictions of performance.]

- Assume that the differences in performance are determined **entirely by the amount of time spent in the closures called by the iterators.**

Under these assumptions, estimate the following quantities for each algorithm:

1. How many addition or subtraction operations are done for each pixel in the image?
2. How many multiplication operations are done for each pixel in the image?
3. How many division or modulus operations are done for each pixel in the image?
4. How many comparison operations (equality, less than, and so forth) are done for each pixel in the image, not forgetting any loop-termination conditions? [*]
5. How many loads are done for each pixel in the image?
6. Of those loads, what fraction hit in the cache?
7. How many stores are done for each pixel in the image?
8. Of those stores, what fraction are to lines that are already in the cache?

If the answers to questions 1–5 and 7 are the same for two different algorithms, the relative performance will be determined only by the cache performance. If the answers to questions 1–5 and 7 are significantly different, you may find that a lot of arithmetic may cost more than a modest difference in the cache-miss rate. (As a rule of thumb, add and subtract cost the same as a load that hits in the cache, a multiply costs a bit more, and division/modulus cost even more. Comparisons vary, but in a well-behaved program a comparison typically costs about the same as an add or subtract.)

op	+ -	muls	divs mods	comps	loads	hit rate	stores	hit rate
180R								
180C								
90R								
90C								

Once you have estimated the expected cost per pixel, please estimate the expected speed of each of the four operations in the table below. Your speed estimate should include the cost of stores as well as the cost of loads.

Your estimate should be a rank between 1 and 4, with 1 being the fastest and 4 being the slowest. If you think two operations will go at about the same speed, give them the same rank. For example, if you think that both column-major rotations will be the slowest and will run at about the same speed, rank them both 3 and rank the other entries 1 to 2.

	row-major	col-major
180 degree		
90 degree		

To complete this problem successfully, you will need to understand the material presented in class and in Chapter 6 of Bryant and O'Hallaron.

Part C (experimental): Measure and explain improvements in locality

This part of the assignment is to be completed after you have a complete, working implementations of `ppmtrans`. Please **measure the speed** of each of the operations in following table:

	row-major	col-major
180 degree		
90 degree		

One problem is that on modern systems, the time spent doing the rotation might be dwarfed by the time spent reading in a large image. So, rather than benchmarking the entire program, you should ***benchmark only the time spent doing rotations***

Specifically, try something like:

```
use std::time::Instant;

/* eliding code to handle command-line arguments */

/* eliding code to read in source image
and create destination array */

// benchmark only the rotation
    let now = Instant::now();
/* eliding some map_row_major or
map_col_major code to do the rotation */

let elapsed = now.elapsed();
eprintln!("{:.2?}", elapsed);
```

Explain your measurements.

- Explain *how* your measurements matched or did not match your predictions from Part B
- Explain *why* if they did not match your predictions.

In order to see any effects, you must use images that are **too large to fit in the cache.** Your fastest rotation should take several seconds; if it does not, you need a larger image.

- You will find a very small supply of large images in `/csc/411/images/large` on `homework`, as well as on EdStem.
- You can create your own large image by using any JPEG file with `djpeg` and `pnmscale`. Experiment until you get something of reasonable size.

Example command lines:

```
djpeg /csc/411/images/large/egrets.jpg |
     target/release/ppmtrans --rotate 90 | display -

djpeg /csc/411/images/large/winter.jpg | pnmscale 1.2 |
     target/release/ppmtrans --rotate 180 > /dev/null

target/release/ppmtrans --rotate 90 --row-major < path/to/blackpoint.ppm > destination.ppm
```

Be sure all your measurements are done with the same image to the same scale

Part D (theoretical): Reason about a memory layout that might perform better

- Particularly for 90-degree rotations, can you come up with a design (a memory layout) that might exhibit *better* cache locality than the one you chose?
- Please don't write a design document, just describe the `Array2` conceptually
- Don't write any code for this

Where to get what

You'll want to create a directory called `locality` to hold your entire project.

You may wish to copy (rather than move) your `array2` project from the previous assignment into that `locality` directory; you'll want to also create the `ppmtrans` project with:

```
cargo new --bin ppmtrans
cd ppmtrans
cargo add csc411_image@0.5
```

If you do not have a working Array2 from assignment 2, you may use the official solution or use one from a classmate. You must give credit to the authors in your README.

Geometric calculations we have done for you

What's important about this assignment is **how locality of memory accesses affects performance,** not how to rotate images. We therefore inform you that we believe:

- If you have an original image of size $w \times h$, then when the image is rotated 90 degrees, pixel (i, j) in the original becomes pixel $(h - j - 1, i)$ in the rotated image.
- When the image is rotated 180 degrees, pixel (i, j) becomes pixel $(w - i - 1, h - j - 1)$.

What we expect from your preliminary submission

Your preliminary submission should include your design work for part A as well as all of part B.

For Part A, please use the **design checklist** (design-pgm.pdf) for writing programs. **We are especially interested in knowing what additional components you plan to use to implement `ppmtrans` and how those components work together to solve the problem.** We expect you to describe a **modular architecture** and to exploit **procedural abstraction.**

Major hint for the design

Again, the checklist is for you. What we want to see are **modules**, **invariants** for your image transformations, and what **subproblems** might be abstracted and reused. You should also highlight some example inputs and outputs (as a diagram or a sketch, for instance). This might help you get your invariants right.

A design that says "given an image, the result will be that image rotated clockwise" will get no credit.

A design that says something like "given a pixel at coordinates (x,y), the resulting image will have that pixel value at coordinates (?,?)," but with something concrete in place of the question marks, will fare much better.

The estimates are also essential

Please submit two text files:

DESIGN(.txt or .pdf or .md) for your design work for Part A. ESTIMATES(.txt or .pdf or .md) for your estimates of locality, work per pixel, and total cost

What we expect from your final submission

Your implementation, to be submitted on Gradescope, should include

- A README.txt or .md or .pdf file which
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken
  - Identifies what has been correctly implemented and what has not
  - Documents the architecture of your solutions.
  - Gives measured speeds for Part C and explains them.
  - Reasons about a possible answer for Part D.
  - Says approximately how many hours you have spent completing the assignment
- Source code for `array2` and `ppmtrans`
  - if you borrowed someone else's `array2`, your README must explain this

You should zip up your submission similarly to past projects.

A reasonable zip command will be something like (while sitting in the parent directory of `locality`):

```
zip -r locality.zip locality -x "locality/*/Cargo.lock"
"locality/*/target/**" "locality/*/.git/**"
```

Avoid common mistakes

Here are the mistakes most commonly made on this project:

- It's a mistake to submit, in place of an invariant, a narrative description of a sequence of events.
- It's a mistake to try to explain a complex invariant in informal English.
- It's a mistake to analyze a rotation experiment if the rotation completes in less than a few seconds.
- When two programs perform very differently, and the programs have very different loop structures, it's a mistake to try to explain performance differences by appealing to locality.

Handling command-line options

You are encouraged to use the `clap` command-line argument processing crate, which is well-documented and used by several major Rust projects. Some information is in the Rust CLI book here: https://rust-cli.github.io/book/tutorial/cli-args.html

You can use `cargo add clap --features derive` inside your `ppmtrans` directory to add the latest version of CLAP to your dependencies.

As an example, consider the following snippet:

```
use clap::Parser;
#[derive(Parser, Debug)]
#[clap(author, version, about, long_about = None)]

struct Args {
    // Flip
    #[clap(long = "flip", required = false)]
    flip: Option<String>,
    // Rotation
    #[clap(short = 'r', long = "rotate")]
    rotate: Option<u32>,
    // Transposition
    #[clap(long = "transpose")]
    transpose: bool,
}

fn main() {
    let args = Args::parse();
    let rotate = args.rotate;
}
```