COMP 40 Assignment: Locality and the costs of loads and stores

Designs and experimental estimates due Sunday, October 2, at 11:59 PM. Full assignment due Thursday, October 6, at 11:59 PM.

Overview and purpose

This assignment is all about the cache and locality. You'll implement **blocked** two-dimensional arrays, which you'll then use to evaluate the performance of image rotation using three different array-access patterns with different locality properties. You'll also see how to write code that is polymorphic in an array type.

The assignment has two parallel tracks:

1. On the *design and building track*, you will implement *blocked* two-dimensional arrays and *polymorphic* image rotation,
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.

As described in Section 2, in this assignment we provide you with a *lot* of code and information. It will take time to assimilate. You can get some of the code by running the commands

```
git clone linux.cs.tufts.edu:/comp/40/git/locality
cd locality
```

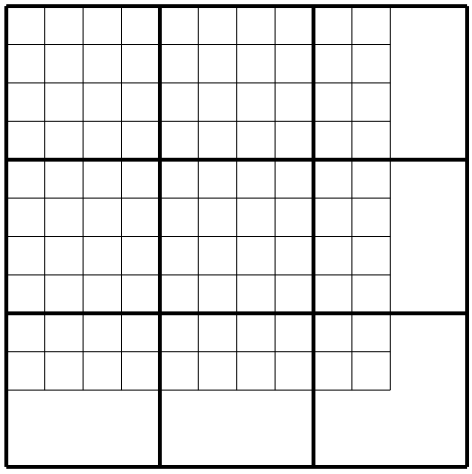
which will create and enter a directory called `locality`. Do this at the start of the assignment.

Contents

1 Problems

1.1 Part A (design/build): Improving locality through blocking

In this part of the assignment, you will implement a standard technique for improving locality: *blocking*. The idea is best expressed in a picture. Here is a 10-by-10 array organized in 4-by-4 blocks:



The idea is simple: the blocked array has a similar *interface* to `UArray2`, but a different *cost model*. In particular,

- **Cells in the same block are located near each other in memory**.
- **Mapping is done by blocks**, not rows or columns. Mapping visits all cells in one block before moving on to the next block.
- **Some memory is wasted** at the right and bottom edges: not all the cells in those blocks are used. But if the array is large, then the wasted memory has size $O(\sqrt{n})$ and is unimportant. If the array is small, it probably fits in the cache and you shouldn't use blocking.

You have just one task for this part:

- **Implement blocked arrays** as described in the `UArray2b` interface below. Use file `uarray2b.c`.

Required interface

Since you have already been through a very similar design exercise, I will not ask you to repeat it. Instead, I am specifying an interface, and I suggest a design which you may use if you wish. The interface you are to implement, to be called `UArray2b`, appears in Figure 1. The `blocksize` parameter to `UArray2b_new` counts the number of *cells* on **one side** of a block, so the actual number of cells in a block is `blocksize * blocksize`. The number of *bytes* in a block is `blocksize * blocksize * size`. The `blocksize` parameter has no effect on semantics, only on performance.

The `UArray2b_new_64K_block` allows you to default the blocksize; it is similar to `UArray2_new`. It chooses a `blocksize` that is as large as possible while still allowing a block to fit in 64KB of RAM. If a single cell will not fit in 64KB, the block size should be 1. On almost any machine built in the last five years, the L1 data cache will hold 128KB of data, so if you create arrays using `UArray2b_new_64K_block`, you can fit two blocks in cache at one time.

```
#ifndef UARRAY2B_INCLUDED
#define UARRAY2B_INCLUDED

#define T UArray2b_T
typedef struct T *T;

extern T    UArray2b_new (int width, int height, int size, int blocksize);
  /* new blocked 2d array: blocksize = square root of # of cells in block */
extern T    UArray2b_new_64K_block(int width, int height, int size);
  /* new blocked 2d array: blocksize as large as possible provided
     block occupies at most 64KB (if possible) */

extern void  UArray2b_free  (T *array2b);

extern int   UArray2b_width (T array2b);
extern int   UArray2b_height(T array2b);
extern int   UArray2b_size  (T array2b);
extern int   UArray2b_blocksize(T array2b);

extern void *UArray2b_at(T array2b, int i, int j);
  /* return a pointer to the cell in column i, row j;
     index out of range is a checked run-time error
   */

extern void  UArray2b_map(T array2b,
    void apply(int i, int j, T array2b, void *elem, void *cl), void *cl);
      /* visits every cell in one block before moving to another block */

/* it is a checked run-time error to pass a NULL T
   to any function in this interface */

#undef T
#endif
```

Figure 1: Interface for blocked arrays

One possible architecture for your implementation

If you wish, you may use your own design and architecture for the implementation of `UArray2b`, or you may use one of mine described as follows:

> Here is a simple architecture for `UArray2b`. Because of the many layers of abstraction, it does not perform very well, but it is relatively easy to implement.
>
> - An `UArray2b_T` can be represented as an `UArray2_T`, each element of which contains one block.
> - A block should be represented as a single `UArray_T`. This representation guarantees that cells in the same block are in nearby memory locations.
> - To find the cell at index (i, j), first find the block at index (`i / blocksize`, `j / blocksize`). Within that block, use the cell at index `blocksize * (i % blocksize) + j % blocksize`.
> - Your mapping function should visit all the cells of one block before moving onto the cells of the next. **Blocks on the bottom and right edges may have unused cells**, and your mapping function must **not** visit these cells.

If you implement this design successfully, it is not too difficult to modify the code such that your blocked array is stored in a single, contiguous area of memory. Once you have the address arithmetic right, you can get a substantial speedup by avoiding all the memory references involved in going indirectly through the `UArray2` and `Array` abstractions. But the focus of this assignment is not on performance, and a faster implementation is purely optional.

My solutions

I have written two solutions to this problem. The one that uses the design sketched above is about 175 lines of C, 50 of which appear at the end of this assignment. I then wrote another, faster solution which is about 130 lines of C. The faster solution has a significantly more complicated invariant and was correspondingly more difficult to get right.

1.2 Part B (design/build): supporting polymorphic manipulation of 2D arrays

You now have two different representations of two-dimensional arrays: `UArray2`, which supports column-major and row-major mapping, and `UArray2b`, which supports block-major mapping. In order not to duplicate code, we want to write image rotations that can operate on *either* kind of array. To achieve this kind of reuse, we resort again to polymorphism: we define an interface `A2Methods` that can represent either kind of two-dimensional array. You write *one* image-rotation program against this interface, and you can use it with *two* implementations.

- Because I have specified the exact interface for `UArray2b`, I can provide an implementation of `A2Methods` that uses `UArray2b`.
- Because you designed the `UArray2` interface yourself, *you* will provide an implementation of `A2Methods` that uses `UArray2`. My implementation is in file `a2blocked.c`; **your implementation goes into** `a2plain.c`.

The `A2Methods` interface uses the same principles as the declaration of an abstract class in a language like C++, C#, Java, or Smalltalk. Instead of calling functions by name, you will call through *pointers* to functions. Those pointers live in a *method suite* of type `A2Methods_T`, which is a pointer to a record of function pointers (Figure 2). For each of these function pointers, you will you will need to create a `static` function that calls into `UArray2`. To implement your method suite, you put pointers to those functions into a `struct`. Looking at `a2blocked.c` will show you a complete example, and in Figure 3 we also provide a template for your `a2plain.c`.

```
typedef void *A2;  // an unknown type that represents a 2D array of 'cells'
typedef void A2Methods_Object;  // an unknown sequence of bytes in memory
                                // (element of an array)

typedef void A2Methods_applyfun(int i, int j, A2 array2,
                                A2Methods_Object *ptr, void *cl);
typedef void A2Methods_mapfun(A2 array2, A2Methods_applyfun apply, void *cl);

typedef void A2Methods_smallapplyfun(A2Methods_Object *ptr, void *cl);
typedef void A2Methods_smallmapfun(A2 a2, A2Methods_smallapplyfun f, void *cl);

typedef struct A2Methods_T { // operations on 2D arrays

  // it is a checked run-time error to pass a NULL 2D array to any function,
  // and except as noted, a NULL function pointer is an *unchecked* r. e.

  A2 (*new)(int width, int height, int size);
    // creates a distinct 2D array of memory cells, each of the given 'size'
    // each cell is uninitialized
    // if the array is blocked, uses a default block size

  A2 (*new_with_blocksize)(int width, int height, int size, int blocksize);
    // creates a distinct 2D array of memory cells, each of the given 'size'
    // each cell is uninitialized
    // if the array is blocked, the block size given is the number of cells
    //    along one side of a block; otherwise 'blocksize' is ignored

  void (*free)(A2 *array2p);
    // frees *array2p and overwrites the pointer with NULL

  // observe properties of the array
  int (*width)    (A2 array2);
  int (*height)   (A2 array2);
  int (*size)     (A2 array2);
  int (*blocksize)(A2 array2); // for an unblocked array, returns 1

  A2Methods_Object *(*at)(A2 array2, int i, int j);
    // returns a pointer to the object in column i, row j
    // (checked runtime error if i or j is out of bounds)

  // mapping functions
  void (*map_row_major)  (A2 array2, A2Methods_applyfun apply, void *cl);
  void (*map_col_major)  (A2 array2, A2Methods_applyfun apply, void *cl);
  void (*map_block_major)(A2 array2, A2Methods_applyfun apply, void *cl);
  void (*map_default)    (A2 array2, A2Methods_applyfun apply, void *cl);
    // each mapping function visits every cell in array2, and for each
    // cell it calls 'apply' with these arguments:
    //    i, the column index of the cell
    //    j, the row index of the cell
    //    array2, the array passed to the mapping function
    //    cell, a pointer to the cell
    //    cl, the closure pointer passed to the mapping function
    //
    // These functions differ only in the *order* in which they visit cells:
    //   - row_major visits each row before the next, in order of increasing
    //     row index; within a row, column numbers increase
    //   - col_major visits each column before the next, in order of
    //     increasing column index; within a column, row numbers increase
    //   - block_major visits each block before the next; order of
    //     blocks and order of cells within a block is not specified
    //   - map_default uses a default order that has good locality
    //
    // In any record, map_block_major may be NULL provided that
    // map_row_major and map_col_major are not NULL, and vice versa.

  // alternative mapping functions that pass only cell pointer and closure
  void (*small_map_row_major)  (A2 a2, A2Methods_smallapplyfun apply, void *cl);
  void (*small_map_col_major)  (A2 a2, A2Methods_smallapplyfun apply, void *cl);
  void (*small_map_block_major)(A2 a2, A2Methods_smallapplyfun apply, void *cl);
  void (*small_map_default)    (A2 a2, A2Methods_smallapplyfun apply, void *cl);

} *A2Methods_T;
```

Figure 2: Polymorphic interface for manipulating two-dimensional arrays

```
#include <stdlib.h>

#include <a2plain.h>
#include "uarray2.h"

// define a private version of each function in A2Methods_T that we implement

static A2Methods_UArray2 new(int width, int height, int size) {
  return UArray2_new(...);
}

static A2Methods_UArray2 new_with_blocksize(int width, int height, int size,
                                            int blocksize)
{
  (void) blocksize;
  return UArray2_new(...);
}

 ... many more private (static) definitions follow ...

// now create the private struct containing pointers to the functions

static struct A2Methods_T uarray2_methods_plain_struct = {
  new,
  new_with_blocksize,
   ... other functions follow in order, with NULL for those not implemented ...
};

// finally the payoff: here is the exported pointer to the struct

A2Methods_T uarray2_methods_plain = &uarray2_methods_plain_struct;
```

Figure 3: Boilerplate for implementing a struct pointer of type A2Methods_T

```
typedef void Array2_apply(int row, int col, void *elem, void *cl);
extern void Array2_map_row_major(Array2_T a2, Array2_apply apply, void *cl);

struct a2fun_closure {
  A2Methods_applyfun *apply; // apply function as known to A2Methods
  void *cl;               // closure that goes with that apply function
  A2Methods_UArray2 array2;   // array being mapped over
};

static void apply_a2methods_using_array2_prototype
               (int row, int col, void *elem, void *cl)
{
  struct a2fun_closure *f = cl; // this is the function/closure originally passed
  f->apply(col, row, f->array2, elem, f->cl);
}

static void map_row_major(A2Methods_UArray2 array2, A2Methods_applyfun apply, void *cl) {
  struct a2fun_closure mycl = { apply, cl, array2 };
  Array2_map_row_major(array2, apply_a2methods_using_array2_prototype, &mycl);
}
```

Figure 4: Mediating between map/apply functions that use different prototypes

The interface you are to implement is defined in `a2plain.h`:

```
#include <a2methods.h>

extern A2Methods_T uarray2_methods_plain; // functions for normal arrays
```

You should **write file `a2plain.c`, which implements this interface**. It should look something like the template in Figure 3. The only tricky bit is resolving differences in your apply functions. Let's suppose that your `UArray2` apply and row-major map functions look like this:

```
typedef void Array2_apply(int row, int col, void *elem, void *cl);
extern void Array2_map_row_major(Array2_T a2, Array2_apply apply, void *cl);
```

This "inner" apply function is compatible with `UArray2`, but it is not the same as the "outer" apply function used in the `A2Methods` interface shown in Figure 2. The exported mapping function will receive an "outer" apply function that is compatible with the `A2Methods` interface, and you will have to create an "inner" apply function that is compatible with your own personal `UArray2` interface.

1. Define a new closure type `a2fun_closure`, which holds the outer apply function and its closure, plus any other information that's expected by the outer apply function but not provided by the inner apply function. In this case, the "other information" is the array.

2. Define a new, "inner" apply function that can be passed to your `UArray2_map_row_major`. This apply function grabs information from the `a2fun_closure`, and it applies the "outer" apply function. In other words, it's just a proxy.

3. Your `A2Methods` version of `map_row_major`, which you'll export a pointer to, builds an `a2fun_closure`, and then calls `UArray2_map_row_major` using the new closure and the inner apply function. The new closure always contains the old closure and the outer apply function.

You can see a full example in Figure 4.

Here is a summary of your obligations for this part:

- You submit a file `a2plain.c` which exports the single pointer `uarray2_methods_plain`.
- In the methods suite, you *must* implement all the methods from `new` through `at`.
- Of the mapping methods, you *must* implement `small_map_row_major`, `small_map_col_major`, and `small_map_default`. For `small_map_default`, you should use either a row-major or a column-major mapping, whichever you think has better locality.
- If you can, you *should* implement methods `map_row_major`, `map_col_major`, and `map_default`. For `map_default`, you should use either a row-major or a column-major mapping, whichever you think has better locality.
- Because `UArray2` does not support blocking, you *must not* implement `map_block_major` or `small_map_block_major`. These pointers must be `NULL`.

1.3 Part C (design/build): ppmtrans, a program with straightforward locality properties

Using the A2Methods abstraction, 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 map_row_major
-col-major
       Copy pixels from the source image using map_col_major
-block-major
       Copy pixels from the source image using map_block_major
```

- You **must not call UArray2 functions or UArray2b functions directly**. Instead you must call *indirectly* through the function pointers in a methods suite.
- You **must use the mapping functions defined in a2methods.h**, *not* nested for loops.
- For row-major and column-major mapping, you will use the methods suite uarray2_methods_plain that you will have created in Part B of this homework. For block-major mapping, you will use the methods suite uarray2_methods_blocked that we provide in interface a2blocked.h.

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.*
>
> *In a blocked representation, the mapping of blocks to blocks is not obvious. To understand the locality properties of blocked array processing, you will have to think carefully.*

My solution to this problem is about 150 lines of code.

1.4 Part D (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 C. Please **estimate the expected cache hit rate for reads** of each of the six 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	blocked access
90-degree rotation			
180-degree rotation			

Your estimate should be a **rank between 1 and 6**, with 1 being the best hit rate and 6 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 5 and rank the other entries 1 to 4.

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.[1]
- Assume that the **differences in performance are determined entirely by the amount of time spent in the mapping functions.**
- Depending on how you design your representation and your mapping algoriths, a mapping function may use one, two, three, or even four nested loops. Assume that **only the operations in the innermost loop** matter.

[1] This assumption oversimplifies the cache's behavior significantly, but it will be good enough to enable reasonable predictions of performance.

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 the loads in question 5, **what fraction hit in the cache**?
7. How many *stores* are done for each pixel in the image?
8. Of the stores in question 7, **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.)

Please submit the *expected work per pixel* in a table like the following:

Kind of rotation	adds/subs	multiplies	divs/mods	compares	loads	hit rate	stores	hit rate
180-degree row-major								
180-degree column-major								
180-degree block-major								
90-degree row-major								
90-degree column-major								
90-degree block-major								

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

	row-major access	column-major access	blocked access
90-degree rotation			
180-degree rotation			

Your estimate should be a **rank between 1 and 6**, with 1 being the fastest and 6 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 5 and rank the other entries 1 to 4.

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

1.5 Part E (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 access	column-major access	blocked access
90-degree rotation			
180-degree rotation			

In detail,

- Do your measurements using the `/usr/bin/time` command, and report the *user CPU time*.
- For blocked access, use 64KB blocks.
- Explain your measurements.

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 `/comp/40/images/large`.
- 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 /comp/40/images/from-wind-cave.jpg | pnmscale 3.5 |
   /usr/bin/time ./ppmtrans -rotate 90 | display -
djpeg /comp/40/images/wind-cave.jpg | pnmscale 1.2 |
   /usr/bin/time ./ppmtrans -rotate 90 | display -
```

- If you need to store a large image, you can create files and directories in the `/data` area, and they will not count against your disk quota. Or you can request that your disk quota be enlarged to 10GB; fill out the form at

  `https://www.eecs.tufts.edu/userguide/forms/quota.php`

  and list me as faculty sponsor.

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

2 Infrastructure that we provide

This section identifies infrastructure you can use for this assignment.

2.1 A polymorphic interface to two-dimensional arrays

1. Figure 2 on page 5 gives a polymorphic interface that describes a method suite for two-dimensional arrays. You will provide a method suite that works with your design and implementation of `UArray2`; we provide an implementation that works with `UArray2b`. For a rather simple example of how to use the 2D-array methods, see sample file `a2test.c` (item 5 below). The file `a2methods.h` appears in `/comp/40/include` and should not be copied.

2.2 Other interfaces we have designed for you

The interfaces below appear in `/comp/40/include`. **Do not copy these files**. You should be able to compile against any of these interfaces by using the option `-I/comp/40/include` with `gcc`.

2. Files `a2plain.h` and `a2blocked.h` define two interfaces that promise method suites. We implement `a2blocked.h`; you should be able to link against it by using the options `-L/comp/40/lib64 -l40locality` with `gcc`.

3. File `uarray2b.h` defines the `UArray2b` interface. (You write the implementation.)

4. File `pnm.h` defines functions you can use to read, write, and free portable pixmap (PPM) files. It defines a representation for colored pixels. The pixmap itself is represented as type `void *`; you will use this code with the `A2Methods_T` methods.

   The `Pnm` interface uses the `A2Methods` interface.

   You should be able to link against our implementation of `pnm.h` by using the options

   ```
   -L/comp/40/lib64 -l40locality -lnetpbm
   ```

   with `gcc`.

2.3 Test code for two-dimensional arrays

5. As usual when implementing polymorphism in C, it is possible to make a mistake with `void *` pointers. You will therefore want to **run small test cases using `valgrind`** in order to flush out potential memory errors. We provide one sample test case in file `a2test.c`; it tests the `cell` and `at` methods as well as row-major mapping, if present. Before you can use it you will need to implement `uarray2b.c` or `a2plain.c` or both.

   Once you can build `a2test`, run `valgrind ./a2test`.

2.4 Other source code

6. We provide C source code for `a2blocked.c`, which you don't need to compile, but you might find useful to study. We also provide incomplete versions of `a2plain.c` and `ppmtrans.c`.

2.5 Where to get what

7. All of the C source we provide for you is in a git repository. That repository also contains a `compile` script that builds `a2test`; you will need to extend the script to build `ppmtrans`. You will want to remember options like

   ```
   sh compile uarray2b.c     # compile just the one .c file
   sh compile -nolink       # compile all .c, but don't link anything
   sh compile -link a2test  # build executable binary a2test
   ```

   You get all these sources by

   ```
   git clone linux.cs.tufts.edu:/comp/40/git/locality
   ```

   which will create a subdirectory `locality`. **We recommend you *begin* the assignment by creating a directory using `git clone`.**

8. Interfaces `uarray2b.h`, `a2methods.h`, `a2blocked.h`, and `a2plain.h` all appear in `/comp/40/include`. **Don't copy these interfaces**.

2.6 Geometric calculations we have done for you

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

9. 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.

10. When the image is rotated 180 degrees, pixel (i, j) becomes pixel $(w - i - 1, h - j - 1)$.

3 What we expect from your preliminary submission

Your preliminary submission should include your **design work** for parts A and C as well as **all of part D**.

- For Part C, please use the design checklist 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**.

- For Part A, please use the design checklist for abstract data types. If we ignore costs, then in the world of ideas, `UArray2b` **cannot be distinguished** from `UArray2`. So all of your test cases and examples carry over from the previous assignment, and you need not repeat them.

  But we expect you to pay special attention to the **representation** and its **invariants**. Please be sure your submission explains

  1. How you will translate cell coordinates (i, j) into a C pointer in *your* representation. (If you use my design, your explanation will probably involve block coordinates—or a block number—and the index of the cell within the block.) The best possible explanation is a *precise* one using a **set of equations**.
  2. How you will translate a location within *your* representation (which in my design would be the combination of block coordinates and the index of a cell within the block) back to pixel coordinates (i, j). The best possible explanation is a *precise* one using a **set of equations**.
  3. What representation you will use for a single block.
  4. What representation you will use for a 2-dimensional array of blocks.

  Please submit two files:

- `DESIGN` for your design work for Parts A and C.

- `ESTIMATES` for your estimates of locality, work per pixel, and total cost

  Submit using `submit40-locality-design`.

4 What we expect from your final submission

Your **implementation**, to be submitted using `submit40-locality`, should include

1. A `README` 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 E and **explains** them.
- Says **approximately how many hours you have spent** completing the assignment

2. The file `ppmtrans.c`.
3. File `uarray2b.c`, which implements the `UArray2b` interface. This file should include internal documentation explaining your representation and its invariants.
4. File `a2plain.c`, which provides a method suite as described by the `a2methods.h` interface.
5. Any other files you may have created as useful components.
6. A `compile` script which when run using

   ```
   sh compile
   ```

   encounters no errors and builds *two* executable binaries: `a2test` and `ppmtrans`.

   - `ppmtrans` should be linked with `ppmtrans.o`, `uarray2.o`, `arrayb.o`, `a2plain.o`, `a2blocked.o`, and probably with other relocatable object files and libraries.

5 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 very different loop structures, it's a mistake to try to explain performance differences by appealing to locality.

6 Code to handle command-line options and choose methods

To deal with command-line options in `ppmtrans.c`, consider the code below. This code does not help you decide if a file has been named on the command line, which determines whether you read from that file or from standard input. To make this decision, you will need to examine the values of `i` and `argc`.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "assert.h"
#include "a2methods.h"
#include "a2plain.h"
#include "a2blocked.h"
#include "pnm.h"

int main(int argc, char *argv[]) {
  int rotation = 0;
  A2Methods_T methods = uarray2_methods_plain; // default to UArray2 methods
  assert(methods);
  A2Methods_mapfun *map = methods->map_default; // default to best map
  assert(map);
#define SET_METHODS(METHODS, MAP, WHAT) do { \
      methods = (METHODS); \
      assert(methods); \
      map = methods->MAP; \
      if (!map) { \
        fprintf(stderr, "%s does not support " WHAT "mapping\n", argv[0]); \
        exit(1); \
      } \
    } while(0)

  int i;
  for (i = 1; i < argc; i++) {
    if (!strcmp(argv[i], "-row-major")) {
      SET_METHODS(uarray2_methods_plain, map_row_major, "row-major");
    } else if (!strcmp(argv[i], "-col-major")) {
      SET_METHODS(uarray2_methods_plain, map_col_major, "column-major");
    } else if (!strcmp(argv[i], "-block-major")) {
      SET_METHODS(uarray2_methods_blocked, map_block_major, "block-major");
    } else if (!strcmp(argv[i], "-rotate")) {
      assert(i + 1 < argc);
      char *endptr;
      rotation = strtol(argv[++i], &endptr, 10);
      assert(*endptr == '\0'); // parsed all correctly
      assert(rotation == 0   || rotation == 90
          || rotation == 180 || rotation == 270);
    } else if (*argv[i] == '-') {
      fprintf(stderr, "%s: unknown option '%s'\n", argv[0], argv[i]);
      exit(1);
    } else if (argc - i > 2) {
      fprintf(stderr, "Usage: %s [-rotate <angle>] "
              "[-{row,col,block}-major] [filename]\n", argv[0]);
      exit(1);
    } else {
      break;
    }
  }
   ...
}
```