

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

Designs and experimental estimates due Thursday, October 8, at 11:59 PM. Full assignment due Tuesday, October 13, 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.

The assignment has two parallel tracks:

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

As described in Section 3, 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	New-technology alert	2
2	Problems	2
2.1	Part A (design/build): <code>ppmtrans</code> , a program with straightforward locality properties	2
2.2	Part B (design/build): Improving locality through blocking	3
	Required interface	4
	One possible architecture for your implementation	5
2.3	Part C (experimental): Analyze locality and predict performance	5
2.4	Part D (experimental): Measure improvements in locality	6
3	Infrastructure that we provide	6
3.1	Test code for two-dimensional arrays	6
3.2	A macro interface to two-dimensional arrays	6
3.3	Interfaces we have designed for you	8
3.4	Wrapper code we provide for <code>ppmtrans</code>	8
3.5	Geometric calculations we have done for you	8
4	What we expect from your preliminary submission	8
5	What we expect from your final submission	9
6	Boilerplate for command-line options	11

1 New-technology alert

The algorithm for image rotation is the same, no matter whether you are using blocked or unblocked two-dimensional arrays. In a normal situation, we would handle this by writing image rotation in a polymorphic style using `void *` and function pointers.¹ However, the number and types of the function pointers in question are extremely involved, and it would be too easy for you to introduce errors. Worse, most errors would not be detectable because typecasts would be required. Instead, we will define a polymorphic abstraction using C preprocessor macros. This technology has two disadvantages:

- It is yet another thing to learn, and you may find it confusing—especially creating macros with arguments. You may also find it difficult to understand the error messages.
- Unlike a `.h` file, an interface defined by a collection of macros cannot be written down in a formal language and checked by the compiler.

These disadvantages are compensated for by two very significant advantages:

- The macros eliminate the need for typecasting between very complicated function-pointer types. As you know, these kinds of casts frequently lead to errors.
- All the types are checked at compile time.

Today's lab is devoted to this technology.

2 Problems

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

Using your `Array2` abstraction, implement program `ppmtrans`, which is modelled on `jpegtran` and 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).

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

¹This technique is the equivalent of using virtual member functions in C++. In fact, virtual member functions are themselves implemented using function pointers.

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
```

- You must use mapping functions, *not* a nested for loop.
- You must write your code as `ppmtrans-body.i`, in such a way that `ppmtrans.c` will compile. As array operations, use the macros defined in `noblock.h` as much as possible. (The row-major and column-major mapping functions are not defined there, but you should find everything else you need.)

Your `ppmtrans` should read a single `ppm` image either from standard input or from a file named on the command line. 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:

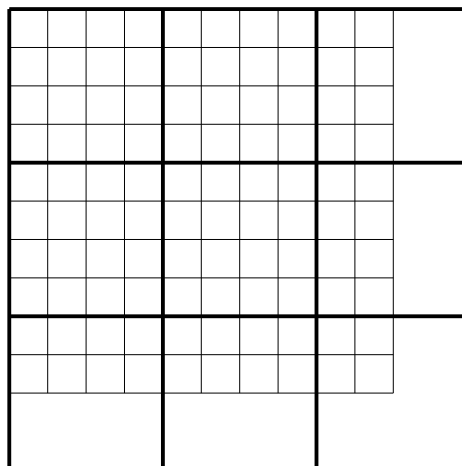
If cells in a row are stored in adjacent memory locations, processing cells in a row has good spatial locality, but it's not clear about processing cells in a column. If cells in a column are stored in adjacent memory locations, processing cells in a column has good spatial locality, but it's not clear 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.

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

2.2 Part B (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 `Array2`, 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; that is, 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 two tasks for this part:

- **Implement blocked arrays** as described in the `Array2b` interface below.
- Using your `array2b.o`, **create a blocked version of `ppmtrans`**; please call the executable binary `ppmtrans-block`.

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 `Array2b`, is as follows:

```
#ifndef ARRAY2B_INCLUDED
#define ARRAY2B_INCLUDED

#define T Array2b_T
typedef struct T *T;

extern T    Array2b_new (int width, int height, int size, int blocksize);
/* new blocked 2d array: blocksize = square root of # of cells in block */
extern T    Array2b_new_64K_block(int width, int height, int size);
/* new blocked 2d array: blocksize chosen so block occupies at most 64KB */

extern void Array2b_free (T *array2b);

extern int  Array2b_width (T array2b);
extern int  Array2b_height(T array2b);
extern int  Array2b_size  (T array2b);
extern int  Array2b_blocksize(T array2b);

extern void *Array2b_get(T array2b, int i, int j);
extern void *Array2b_put(T array2b, int i, int j, void *elem);

extern void Array2b_map(T array2b,
    void apply(int i, int j, T array2b, void *elem, void *cl), void *cl);

#undef T
#endif
```

The `blocksize` parameter to `Array2b_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 `Array2b_new_64K_block` function has the same prototype as `Array2_new`. It chooses a `blocksize` that is as large as possible while still allowing a block to fit in 64KB of RAM. 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 `Array2b_new_64K_block`, you can fit two blocks in cache at one time.

One possible architecture for your implementation

If you wish, you may use your own design and architecture for the implementation of `Array2b`, or you may use one of mine described as follows:

Here is a simple architecture for `Array2b`. Because of the many layers of abstraction, it does not perform very well, but it is relatively easy to implement.

- An `Array2b_T` can be represented as an `Array2_T`, each element of which contains one block.
- A block should be represented as a single `Array_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 $(i / \text{blocksize}, j / \text{blocksize})$. Within that block, use the cell at index $\text{blocksize} * (i \% \text{blocksize}) + j \% \text{blocksize}$.
- 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 clone and 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 `Array2` 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.

2.3 Part C (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 speed** 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 fastest and 6 being the slowest. If you think two operations will operate 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.

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

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

2.4 Part D (experimental): Measure improvements in locality

This part of the assignment is to be completed after you have working implementations of `ppmtrans` and `ppmtrans-block`. 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 `time` command, and report the *user CPU time*.
- For blocked access, use 64KB blocks.

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 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 |  
time ppmtrans -rotate 90 | display -  
djpeg /comp/40/images/wind-cave.jpg | pnmscale 1.2 |  
time ppmtrans -rotate 90 | display -
```

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

3 Infrastructure that we provide

This section identifies infrastructure you can use for this assignment. Where we provide source code, you can get 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`.**

3.1 Test code for two-dimensional arrays

1. Files `a2test.c`, `a2test-block.c`, and `a2test-body.i` test the array put and mapping operations. Using macros you have one body of code (`a2test-body.i`) that is used with both blocked and unblocked representations. The macro interface is described in Section 3.2 below.

3.2 A macro interface to two-dimensional arrays

Our code is intended to work with both `Array2` and `Array2b` interfaces. To make our code polymorphic without losing the benefits of compile-time type checking, we use a set of macros based on the `Array2b` interface. These macros are describe in Figure 1. For examples of their use, please see the `a2test-body.i` file that we provide.

The macro definitions for the `Array2b` interface given above are shown in Figure 2. These definitions are included in files we provide:

A2	Type of a two-dimensional array (either <code>Array2_T</code> or <code>Array2b_T</code>)
<code>anew(w, h, s)</code>	Create a new array of type A2 with the given width, height, and size. If the array is an <code>Array2b_T</code> , the blocksize will be chosen so that a block fits in 64KB.
<code>anew_blocksize(w, h, s, bs)</code>	Create a new array of type A2 with the given width, height, size, and blocksize. If the array is an <code>Array2_T</code> , the blocksize is ignored.
<code>afree(pa)</code>	Free <code>*pa</code> .
<code>aget(a, i, j)</code>	Get a pointer to element (i, j) from array <code>a</code> of type A2.
<code>aput(a, i, j, e)</code>	Write into element (i, j) of array <code>a</code> of type A2.
<code>amap(a, app, cl)</code>	A map function appropriate to the A2 type.
<code>awidth(a)</code>	Return the width of <code>a</code> .
<code>aheight(a)</code>	Return the height of <code>a</code> .
<code>asize(a)</code>	Return the size of <code>a</code> .
<code>ablocksize(a)</code>	Return the blocksize of <code>a</code> , or if <code>a</code> is not blocked, return 1.
<code>amapfun</code>	Type of a map function for A2.
<code>aapply</code>	Type of an apply function for A2.

Figure 1: Macros for two-dimensional arrays

```

#ifndef BLOCK_INCLUDED
#define BLOCK_INCLUDED

#include "array2b.h"

#define A2    Array2b_T

#define anew      Array2b_new_64K_block
#define anew_blocksize Array2b_new
#define afree     Array2b_free

#define aget      Array2b_get
#define aput      Array2b_put

#define awidth    Array2b_width
#define aheight   Array2b_height
#define asize     Array2b_size
#define ablocksize Array2b_blocksize

#define amap Array2b_map
typedef void (*aapply)(int i, int j, A2 a, void *elem, void *cl);
typedef void (*amapfun)(A2 a, aapply f, void *cl);

#endif

```

Figure 2: Macro definitions for *blocked* arrays

- Files `block.h` and `noblock.h` define preprocessor macros which allow you to write array functions that are agnostic about the representation of arrays. Observe how they are used in `a2test.c` and `a2test-block.c`, and how the reusable code in `a2test-body.i` is shared. We expect you to share your implementation of `ppmtrans` in the same way.

You will be able to use `block.h` unchanged, because it is designed for the `Array2b` interface that you are required to implement. You may, however, have to change `noblock.h`, because it is designed for use with the *instructor's* `Array2` interface, and you may have to adapt it to your own `Array2` interface. In particular, you may have to define macros with *arguments*. We suggest that you study the macros that Hanson defines in his `Mem` interface, and that you ask the course staff for help during lab.

3.3 Interfaces we have designed for you

The interfaces below appear in `/comp/40/include`. **Do not copy these files.**

- File `array2b.h` defines the `Array2b` interface.
- 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 both `Array2_T` and `Array2b_T` (see below).

The definition of `struct Pnm_arrayfuncs` contains many pointers to functions. These are probably new to you; they are the programming idiom that C uses instead of the “virtual member functions” (properly known as “methods”) used in C++ classes. There is nothing inherently unsafe about function pointers, but my definition of `struct Pnm_arrayfuncs` uses many `void *` pointers, which we know are **unsafe**. You will therefore want to **run small test cases using `valgrind`** in order to flush out potential memory errors. Start by building `a2test` and then running `valgrind a2test`.

You should be able to compile against `pnm.h` by using the option `-I/comp/40/include` with `gcc`.

You should be able to link against `pnm.h` by using the options `-L/comp/40/lib64 -l40locality -lnetpbm` with `gcc`.

3.4 Wrapper code we provide for `ppmtrans`

- File `ppmtrans.c` is a wrapper that will `#include` your `ppmtrans-body.i` (see below) in a context that uses the existing `Array2` interface from the homework solutions.
- File `ppmtrans-block.c` is a wrapper that will `#include` your `ppmtrans-body.i` (see below) in a context that uses the `Array2b` interface. Notice the initialization of a method suite of type `struct Pnm_arrayfuncs`.

3.5 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

- 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 $(w - i - 1, h - j - 1)$.

4 What we expect from your preliminary submission

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

- For Part A, 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 component work together to solve the problem**. We expect you to describe a **modular architecture** and to exploit **procedural abstraction**.
- For Part B, please use the design checklist for abstract data types. If we ignore costs, then in the world of ideas, `Array2b` **cannot be distinguished** from `Array2`. 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 B.
- ESTIMATES for your cost estimates.

Submit using `submit40-locality-design`.

5 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 D and **explains** them.
 - Says **approximately how many hours you have spent** completing the assignment
2. The file `ppmtrans-body.i`, which will be used to compile both `ppmtrans.c` and `ppmtrans-block.c`.
3. A `compile` script which when run using

```
sh compile
```

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

- `ppmtrans` should be linked with `ppmtrans.o` and `array2.o`, and probably with other relocatable object files and libraries.

- `ppmtrans-block` should be linked with `ppmtrans-block.o` and `array2b.o`, and probably with other relocatable object files and libraries.
4. File `array2b.c`, which implements the `Array2b` interface. This file should include internal documentation explaining your representation and its invariants.
 5. Any other files you may have created as useful components.

6 Boilerplate for command-line options

To dealing with command-line options in `ppmtrans-body.i`, 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 <string.h>
#include <stdlib.h>
#include "assert.h"
#include "pnm.h"
#include "array2b.h"
#include "array2.h"

int main(int argc, char *argv[]) {
    int rotation = 0;
    void (*map)(A2 pixmap, void apply(int, int, A2, void*, void*), void *cl) = amap;

    Pnm_arrayfuncs methods = map != (amapfun) Array2b_map ? Pnm_array2funcs : a2bfuncs;
    assert(methods);
    int i;
    for (i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "-row-major")) {
            if (map != (amapfun) Array2b_map) { // assignment won't segfault
                map = (amapfun) Array2_map_row_major;
            } else {
                fprintf(stderr, "Blocking version of %s does not support row-major mapping\n",
                    argv[0]);
                exit(1);
            }
        } else if (!strcmp(argv[i], "-col-major")) {
            if (map != (amapfun) Array2b_map) { // assignment won't segfault
                map = (amapfun) Array2_map_col_major;
            } else {
                fprintf(stderr, "Blocking version of %s does not support col-major mapping\n",
                    argv[0]);
                exit(1);
            }
        } 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>] [filename]\n", argv[0]);
            exit(1);
        } else {
            break;
        }
    }
    ...
}
```