HW 3 Locality, Part C: Estimates
Feiyu Lu & Jeanne-Marie Musca

A. NUMBER OF HITS

Assumptions

1. The UArray2 implements the 2d array using a single 1d array, which is row-major (although the results should be the same if it is implemented as rows of 1d arrays within a 1d array, given that the 1d array is unboxed and thus allocates memory for the individual row arrays contiguously).

2. To rotate: get the pixel's (col, row) location at time of access and perform calculations to get it's new location.

Table

(1-best/6-worst)	**row-major access (UArray2)**	**col-major access (UArray2)**	**blocked access (UArray2b)**
90-degree rotation	2	5	2 (or 3)
180-degree rotation	2	5	2 (or 3)

Justification:

90-degree rotation vs 180-degree rotation:

Either degree of rotation should have the same number of hits when in the same condition, because we access each element in the same sequence; only the calculation is different once the element has been accessed.This also means that there is bad temporal locality because the same variable is not accessed repeatedly, but it is equally bad in all cases.

So, we will only distinguish among the 3 ways of access:

1. *row-major access in UArray2:*

The access of memory is in the same sequence as the array implementation, so the elements are always contiguous. The hit rate is high due to good spatial locality!

This is a close to ideal situation. The reason we give it score 2 instead of 1 is that the image is larger than the cache, which will cause evictions.

2. *col-major access in UArray2:*

The 2d array is implemented using a 1d array that is row major. So, each element that is accessed will be the whole width of the image away from the previous element in memory (ie, stride k reference pattern where k = width of image). That means that any data that has been "read-for-write" added to the cache might not contain the given element (depending on the size of a line), and if it does, it probably won't contain the next element, which is another whole width away. In other words, this function has poor spatial locality.Thus, there will be a considerably low hit rate when using col-major access.

We give 5 instead of 6 because the locality is not as bad as it would be with RANDOM access. For example, when the first element in column 1 is loaded, the 1st element in column 2 (or even 1st element in later columns, if cache's block size is big) might already be stored in cache.

3. *block-major access in UArray2b:*

UArray2b uses a 2d unboxed array (implemented as a 1d unboxed array). Each entry in the 2d array is a smaller 1d array, representing the blocks. Because the 2d array is unboxed, the 2d array allocates

memory for each smaller 1d array within it. So, all the smaller 1d arrays in the 2d array should be in contiguous parts of memory. The blocks are stored in a 2d array, however, so there are two different orders in which the blocks could be accessed, row major or column major:

a. **row major access of blocks:** the situation will be exactly the same as row-major access, since each block is a continuous 1D array, and blocks are aligned in row-major in an unboxed 2d array. So, there is good spatial locality.

b. **column major access of blocks:**, there will still be a higher number of hits than column major access, because each block in the Array2 will have elements that are contiguous. So, there will be good spatial locality within the blocks, but not across the blocks.

B. OPERATIONS

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.
- 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 algorithms, a mapping function may use one, two, three, or even four nested loops. Assume that only the operations in the innermost loop matter.

Table: expected work per pixel

	adds/subs	multiples	divs/mods	compares	loads	hit rate (1- hi, 6-lo)	stores	hit rate (1- hi, 6-lo)
180-degree row-major	4	2	0	6	12	1	3	1
180-degree column-major	4	2	0	6	12	1	3	1
180-degree block-major	4	2	0	6	12	1	3	1
90-degree row-major	4	2	0	6	12	1	3	1
90-degree column-major	4	2	0	6	12	1	3	1
90-degree block-major	4	2	0	6	12	1	5	1

Justification

90-degree rotation vs 180-degree rotation:

Either degree of rotation should have the same number of hits when in the same condition, because we access them through the same mapping function and then perform the relevant calculations

col-major, row-major & block major

If we look only at the inner loop of each function, each of these mapping functions turn out to be roughly the same.

For each **col-major, row-major** mapping, there are two for-loops, outermost one iterates through the major value, and innermost iterates through the minor value. The mapping functions perform essentially the same function, just with the loops exchanged.

While there is one additional for loop in the **block-major** function, and an additional load for each array that represents a block, we are only concerned with the inner loop which is almost the **same**, as is illustrated by the pseudo-code below:

Pseudo-code for block-major:

```
1. Iterate through blocks in UArray2
iterate through UArray2 row (for loop)
        iterate through UArray2 col (for loop)
                get the array representing the current block

 2. Iterate within block:
                Iterate through UArray (for loop) ← INNERMOST FOR LOOP
                        Apply the function to the pixel
```

Here is a rough break down of operations per subfunction

The sample code below is from row-major mapping

For mapping function: (in the inner loop)

```
for(int r = 0; r < uArray2->height; r++) {
    for(int c = 0; c < uArray2->width; c++) {
            apply(c, r, uArray2, UArray2_at(uArray2, c, r), cl);
    }
}
```

- add/subs: 1 addition
- compares: 1 compare
- loads: 2 function **(APPLY),** 2 variables (width, c)
- hit rate: close to 100% [good temporal locality, variables used each time in loop]
- stores: store c iterator
- hit rate: close to 100% [good temporal locality variable used each time in loop]

For apply function: implemented by client (rotation)

- loads: 1 function **(AT),** 4 variables (c, r, at, uArray2, cl)
- hit rate: close to 100% [good temporal locality, each variable is called each time in loop]
- stores: store the result of uArray_at function call
- hit rate: close to 100% [good temporal locality, stored in a variable defined in the apply function]

For UArray2_at function:

```
return (void *)UArray_at(uArray2->uArr, convertIndex(col, row, uArray2->width));
```

- add/subs: 1 addition
- compares: 5 for error checking
- loads: 1 function **(UArray_at),** 2 variables (width, c)
- hit rate: close to 100% [good temporal locality]
- stores: store c iterator
- hit rate: close to 100% [good temporal locality]

For UArray_at function:

```
return uarray->elems + (i * uarray->size)
```

- add/subs: 1 addition
- multiples: 1 multiplication
- loads: 3 variables (elem, size, i)
- hit rate: close to 100% [good temporal locality because these variables are used repeatedly]

For convertIndex

```
return row * width + col;
```

- add/subs: 1 addition
- multiples: 1 multiplication

C. Speed Estimate

	row-major access	column-major access	block-major access
90-degree rotation	2	4	3
180-degree rotation	2	4	3

Justification

block-major access

Block-major access has roughly the same spatial and temporal locality (therefore same number of cache hits) as row-major access. Still, although block-major access has about the same number of operations as the other two functions if we're just looking at an individual pixel, there is slightly more overhead because of the additional loop, and having to access the block array inside the the second loop for each block, thus, we predict that it will be slightly slower than row-major access, but faster than column-major.

row-major access

Row major access has both high cache hits and has the least number of operations looking at the whole program (roughly equal to column-major access). Thus, we predict that it should be perform the fastest out of the three functions.

column-major access

Column major access has low cache hits (slow) because of bad spatial locality but the same number of calculations and the same number of loops as row-major access(quick). So, we predict that it will be the slowest, but not too bad.