6.172 Project 1
Team Members – Natalia Suarez, Adriano Hernandez

Design Overview

For context on our original rotation algorithm read the beta writeup. For the final version, we tried various different algorithms and either saw no performance improvement, or were unable to finish in time. We saw no performance improvement for tiling (which we implemented twice: recursively once, and iteratively once), and were unable to complete a promising algorithm which we dubbed the "windmill." We also ran out of time implementing our previous algorithm, but with larger, 128x128, blocks. We did not find a more clever way to implement our base 64x64 algorithm either.

Our tiling algorithms simply re-order the way in which we rotate 64x64 sized blocks. They did not have bigger buffers or any other meaningful changes. Perhaps it would have been worth trying tiling in which each tile received an entire buffer onto itself.

We thought tiling would be helpful, since it would have given us tools to minimize the vertical (columnar) distance between any two blocks that would be copied within a short time of each other. Intuitively, it would have potentially sacrificed some of the caching benefits of copying directly along rows in the top left and bottom right quadrants, but by going down a smaller amount of columns, it would have ensured that the top left and bottom right copies are not exclusively traversing columns. Whereas previously the cache would be hit consecutively many times, followed by many consecutive misses, many consecutive hits, and many consecutive misses again, tiling should have a slightly lower miss rate while increasing the rate of interleaving between hits and misses.

Recursive Tiling

In recursive tiling, we picked a tiling threshold under which regular rotation was used. This threshold was measured by height in numbers of blocks. Our tiles were roughly squares but could have also been small rectangles. In this tiling scheme, we tile the top left, top right, bottom left and bottom right. In each of those, we repeat the same process until the height of the tile is under the threshold. At that point we simply use our old 64x64 rotation algorithm to rotate the blocks into place.

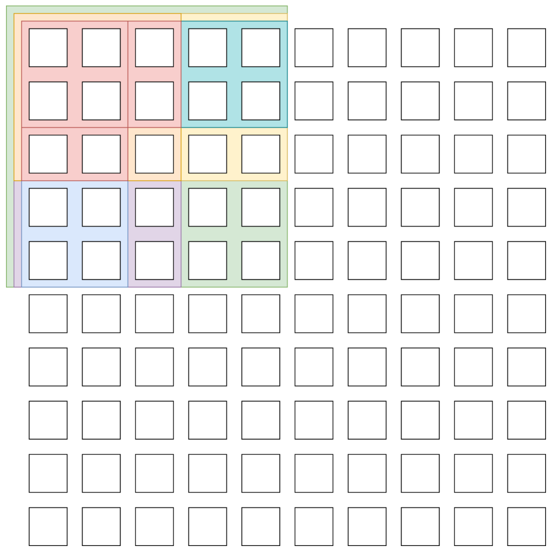
This call-order ensures that our tiles would be in an order from left to right and top to bottom, emphasizing left to right, but in a DFS order. This means that the leftmost, topmost tile would have been copied first, then its tile on the right, then its tile on the bottom left, then its tile on the bottom right. Then the tiles to the right of these four tiles would have been copied also in that order. This is as opposed to iterative tiling in which the tiles go row by row. This means in iterative tiling we copy the top left tile, then the next tile to the right, and so on, until we exhaust the row and copy the next one.

Note that within a tile, everything is copied row by row. That is to say, we copy the top left block, then the block to the right, and so on, until we exhaust the row of blocks inside the tile. Then, we copy the next row, and so on.

For our threshold value we picked 2, 3, 4, and 16. None of them performed better than our original row by row algorithm.

Below is a rough diagram illustrating a potential call order for the recursive tiling algorithm on the upper left quadrant of an image of blocks. The tiling threshold is 2. All of the green is run. This decomposes into first orange, then yellow, then purple. Orange decomposes into red, for which the top left, then the top right, then the bottom is run. Then the orange bottom right runs. Next was yellow, which decomposes into cyan (runs first) and then the remaining yellow. Purple runs next. It decomposes into blue (runs first) followed by the leftover purple. Finally, green runs.

Each run copies the blocks within the corresponding tile. The tiles which have no sub-tiles are those within which blocks are copied row by row (as in the original algorithm). Therefore, blocks inside the upper left red tile copy row-wise.



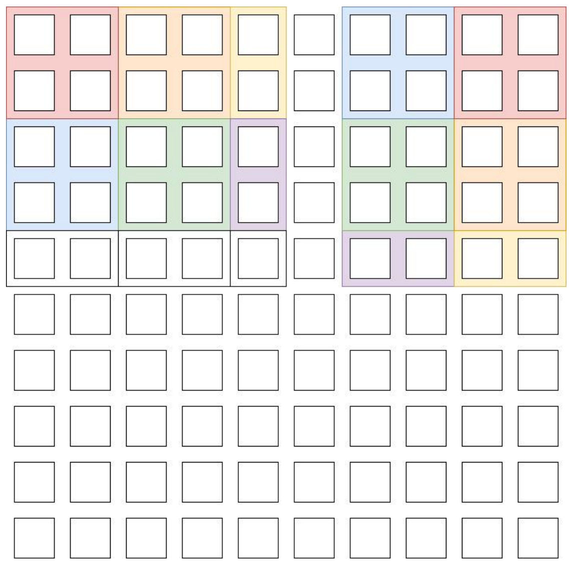
Iterative Tiling

In iterative tiling, we picked a tiling width and height. We then greedly go through tiles from top to bottom and left to right in the upper left rectangle/square until we can no longer do so.

The leftover blocks can be naively rotated or tiled to maximize their size. This choice did not make a difference. This leftover tiling involved picking the tile width (if possible) for the tiles at the bottom, but squashing the height to be as tall as possible, which is necessarily less than the tile height due to the greedy algorithm. Analogously, on the right blocks we squashed the tile width, but kept the tile height. If there was a small square of blocks on the bottom right of the top left hand corner of the image, we tiled them with the side-lengths of that square, which is equal to the minimum height and width of the previously mentioned leftover tiles respectively.

For iterative tiling we picked tile sizes of 2x2, 3x3, 4x4, 16x16 and 2x3 but none of them performed better than our row by row algorithm.

The diagram below illustrates the order of iterative tiling. This example has 2x2 tiles. The elements on the top left quadrant are rotated in tiles. The tiles are colored and will be rotated in the order: red, orange, yellow, blue, green, purple, etcetera. On the top right quadrant the corresponding placement locations are shown in the corresponding colors. Of course, the bottom two quadrants would receive blocks as well, but we have omitted those details for brevity. Note that in this case, we have chosen to create maximally sized tiles on the leftover blocks (yellow, purple, singleton and rectangles at the bottom of the top left quadrant). Also recall that as in the other algorithm, within a tile rotation is row by row.



Windmill

The windmill algorithm is called the “windmill” algorithm because for odd side length (counted in blocks) images it must rotate a cross of blocks in the middle that looks like thin windmill sails.

The goal of this algorithm is to maximize row movement and thus be friendly to cache. It is meant to be used with a big base case (probably using our original row by row rotator) since it is asymptotically inferior to it, moving each block O(logN) (for an NxN matrix) times instead of O(1) times.

The naive windmill algorithm works like this: split the image into four quadrants such that if the image is odd, there is a cross of 64x64 sized blocks down the middle separating four equally sized “quadrants.” Copy the rows of the first quadrant into the rows of the second quadrant, the second into the third, and so on. There are no columnar movements: row in top left goes to row in top right goes to row in bottom right goes to row in bottom left. It is trivially vectorizable. After rotating these rows (or interleaved with them) the cross in the middle (if it exists) should be rotating using the original algorithm. It is probably better to rotate the cross block by block, picking each block just after rotating the 64 corresponding rows to the left, since the cache might already hold that block. Finally, recursively call this algorithm on each of the four quadrants.

The algorithm is correct because of the block rotation argument given to us in the assignment. It is easy to see for matrices whose sizes are powers of two. If we have four quadrants, we rotate each of them and then place it in the correct location, then we rotate the matrix. Similarly, we can reverse the order: move and then rotate. They are equivalent, but in our conception of the windmill algorithm we pick the latter. If we have different sized blocks, because the geometric structure of four quadrants separated by a cross is rotationally symmetric, we can trivially move elements into their positions without worrying about overwrites.

For any word in a row, it must be moved across to another quadrant, and then again for the quadrant within that quadrant and so on, each time dividing the quadrant side length by approximately two. This is why each word moves around O(logN) times instead of O(1) times for an NxN matrix. However, this algorithm is theoretically very good for caching since it exponentially decreases the vertical (columnar) distance between any two blocks which must be rotated in each recursive call. That is because in rotating the quadrants (after moving them), the quadrants’ heights will be half that of the parent. Similarly, the quadrants of the quadrants will have a quarter the length at most, and so on, halving each recursive call.

Thus, if we run it up to some constant depth and within that depth use row by row or tiled rotation, our row by row or tiled rotation may experience a significant cache hit rate and thus be faster. Note that it would be necessary to make a slight modification to the original rotation algorithm. Instead of rotating blocks across quadrants, they'd move across the quadrants of the current rectangle within the image (generated by the recursive process of breaking quadrants into quadrants and so on).

As I mentioned, this algorithm is also trivial to vectorize for row-copying, since there is no columnar movement. It is unfortunate we did not have time to implement this algorithm. We do not have a diagram since we did not implement it.

Bigger Blocks

For bigger blocks we chose 128x128 since it is easier to program. We would have loved to have tried 256x256 for full use of the AVX registers, but we ran out of time. The algorithm remained almost the exact same as for 64x64 size blocks. We picked the top left rectangle (or square) and for each block rotated it and its corresponding three other blocks into their final positions as in the.

For the 128x128 case, we simply greedily took blocks of that size from the top left going right and down until we could no longer fit any more. Because the upper left rectangle (or square) has either an even or an odd side length on each side, there is at most one additional column of 64x64 bit blocks on the right (after the last 128x128 block horizontally) and one additional row of 64x64 bit blocks on the bottom (after the last 128x128 block vertically). Thus, we simply iterate through those after moving the blocks, treating them as in the 64x64 bit case.

As this is similar, visually, as tiling with 2x2-size blocks in which those big tiles are moved jointly, we do not provide another diagram. Refer to iterative tiling for what this would look like.

64x64 Algorithm Speed Up

We also tried speeding up our base 64x64 implementation by calculating endianness in the first row shift to get rid of the extra for loop we were currently utilizing to convert the entire image at once. We also tried to change the order that we were storing blocks of 64x64 into the cache by going in the counterclockwise direction to be more cache friendly but none of these implementations gave us a significant speed up.

MITPOSSE Comments

Most of the comments given to us by our MITPOSSE Deputies were around organization and styling. We addressed all of these concerns by getting rid of extra files and folders that we had and adding comments in our code to better explain methods and method parameters. Our deputies also pointed out some C syntax changes to implement such as the deletion of global variables and the standardizing of type variables across our code (for example not using int in for loops but uint32_t or equivalent).

Help Acknowledgment

We want to thank the 6.172 teaching staff for help with debugging and algorithm questions and general algorithm questions, but also our classmates as we drew inspiration from their work most notably rank 0's dive-and-conquer algorithm.

Work Log

		Duration		
		Adriano	Natalia	
September 15th	2:00 PM	1	1	Outlined our thoughts on the team contract
September 15th	7:00 PM		2	Created team contract
September 16th	8:00 AM	1	1	Read the handout carefully and wrote down some reminders.
September 16th	8:00 PM	2	2	We thought about how to quickly do 64x64 rotations and designed an algorithm.
September 17th	8:00 AM	1		Worked on outer rotation block-rotation
September 19th	5:00 PM		2	Worked on inner 64x64 rotation on simple algorithm described in class
September 20th	8:00 AM	2		Worked on creating a backup slower version of the inner 64x64 rotation
September 20th	8:00 PM		1	Worked on updating algorithm to use new divide and conquer algorithm from recitation
September 21st	4:00 PM		2	Finished implemeting the new divide and conquer algorithm but was not working
September 21st	8:00 AM	2		Worked on the backup 64x64 rotation; did not fix, but added a ton of asserts.
September 21st	6:00 PM	2		At OH getting help with 64x64. Added various tests in my stash branch (adriano/stash).
September 21st	7:30 PM		2	At OH, added endian swap to divide and conquer algorithm which fixed most bugs
September 21st	11:00 PM	2		Fixed some bugs and added more tests. The tests are not exercised. Worked on block rotates.
September 22nd	5:30 PM	2.5	2.5	Debugged our implementation at OH. Went from buggy to Tier 39.
September 22nd	10:00 PM		2	Worked on beta write up
September 26th	5:30 PM	2.5	2	Adriano: worked on recursive tiling; Natalia: looked at rank coding to draw inspiration; Both: thought through algorithms together
September 27th	5:30 PM		4	Started and finished implementing 128 solution with debugging help at OH
September 28th	7:30 PM	4		Worked on recurisve tiling and windmill 128 pseudocode with and without tiling
September 29th	8:15 AM	2.5		Working on iterative tiling, vectorization, memcpy, 128, row-copier (not fully tested).
September 29th	11:30AM	2		Debugged iterative tiling and new indexing scheme
September 29th	3:00PM	6	9	Adriano: continued debugging implementations and got them to work; Natalia: tried to vectorize and unroll loops and speed up code while also implementing a 128 solution and cleaning up code