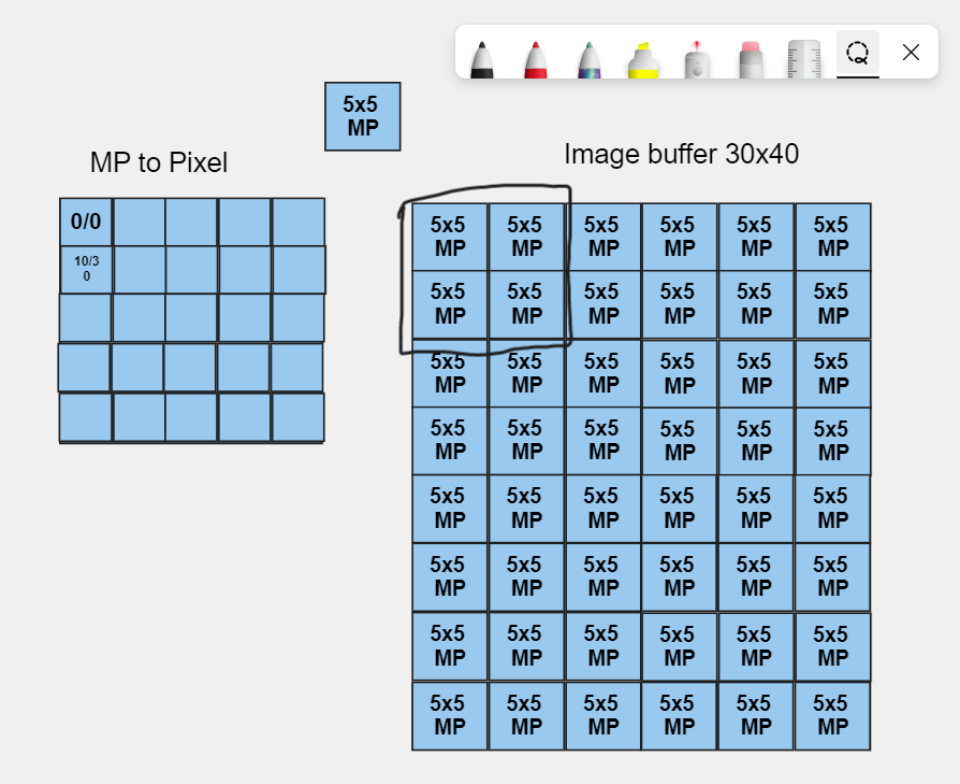
Objective:

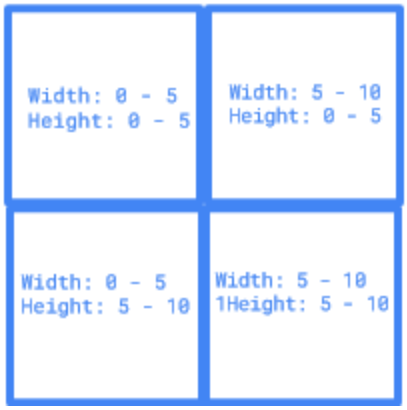
To copy the contents of ‘n’ macropixels onto an image buffer of a particular size. In our case, we are attempting to copy the contents of 48 macropixels(of resolution 5x5) onto a 30*40 image buffer.



Approach:

The 30*40 image buffer would first have to be downsized to 12 10*10 arrays. Each 10*10 array would further be composed of four 5*5 matrices. The first task would be to copy the data from the very first 5*5 matrix onto the start location of the image buffer. This is relatively simple because we would be starting from the (0,0) location for both arrays. But to reduce the number of consecutive copy operations, we can make use of 4 macro pixel units as one super-pixel (that way we would have 12 super-pixels totally, one thread dedicated to each; each super-pixel will have a size of 100). The first macropixel will occupy the first 25 positions, the next will occupy the next 25, until all 100 positions are filled and the super-pixel is populated. Once the lgic is intact, we can directly populate the super-pixels without having to fill the macro pixels individually and copy onto the image buffer.

Below is an example of the first super pixel that will occupy the image buffer.

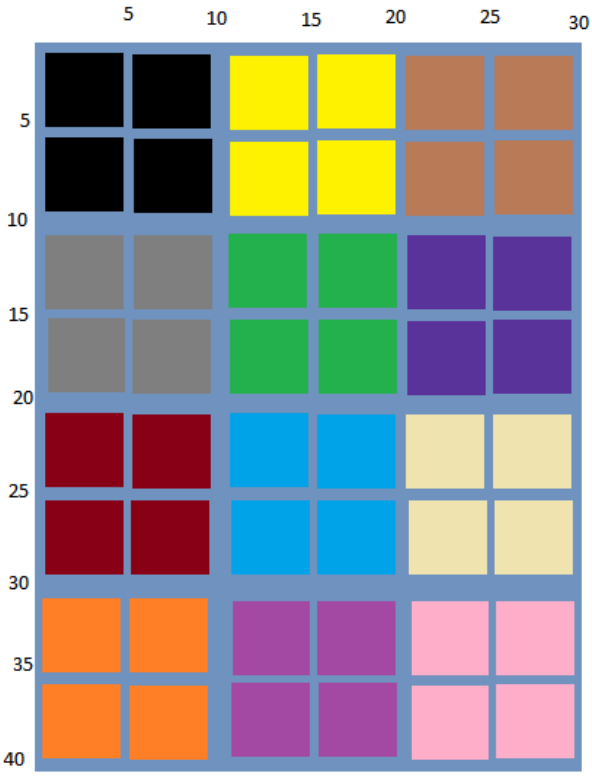


As visible, it is comprised of 4 macropixels, thus having an overall size of 100 pixels.

Once a super-pixel is ready to be flushed, we send a signal to the primary thread of the image buffer to start the copy operation. The same step will have to be repeated 12 times. One thing to note here is that the positions of the next super-pixel might not be sequential (i.e starting pos of int_buff1 is (0,0), but it need not be (10,10) for the next one. Most likely it will be (0,10) and the last element of int_buff2 would be at (10,20). Coordinates here are with respect to the image buffer.) I will have to code in such a way that the position within the super-pixels match their respective positions on the image buffer to ensure that the pixels retain their original position and the final image does not get corrupted.

We are statically allocating memory so that only the required amount of space is consumed.

I have filled in values within the macropixels by simply incrementing the value of a counter variable. The initial plan was to have 12 threads, each assigned to one super-pixel. The thread from each super-pixel will be responsible for copying data from each macropixel and writing the same onto their respective places. Then the entire image buffer will further have another main thread, responsible for further copying the data from each super-pixel onto their respective positions in the image buffer. However, this means that the entire copy operation will go through twice. The macropixels will be copied onto the super-pixel and then the same would be copied into the image buffer. Instead, I'm looking into how to use threads to copy directly from each macropixel onto the image buffer. This` would mean that there has to be individual threads that are isolated from each other, allocated to each macropixel. These threads will further need to be able to be accessed by the main thread of the image buffer.



Here, each colour represents one thread.

The macropixels were available to be filled up into the super-pixel because of global declaration. However, it is required that the contents of one macropixel are not accessible to another macropixel for purposes of data privacy. So we need to declare the macro pixel arrays within their respective thread functions and make sure that these thread functions can communicate with the main thread.

For the main thread (super-pixel in our case) to have access to the macropixel arrays, each macropixel needs to have a shared memory location with the main thread. One approach is to have the super-pixel behave as a parent process and the macropixels behave as child processes. This way, we can have a shared memory space between the two and easily perform the copy operation (perhaps a pipe can be used for data transfer).

With regards to having individual buffers for each thread, the whole process of generating a shared memory space between said buffer and main thread seems cumbersome. To counter this, a global structure method with a common buffer can be used. This common buffer will be used by all threads but only one thread can use it at a time. This can be ensured by making use of a mutex lock on the common buffer. An alternative to even this was making use of mmap. Normally, mmap is used as a shared memory space between the kernel and user space; In this context it can be used between a child and parent process. But this would deem our initial logic of threads null and void.

The final approach that has been decided upon is to directly perform the copy from the super-pixel onto the image buffer with the help of a mutex lock. The copy function is invoked by each thread and the mutex lock ensures that no other operation can take place on it simultaneously.

A nominal testing methodology would be to flip the contents of the image buffer and see if the image itself has been inverted (done by taking the transpose of the matrix and interchanging the positions of the first column with last column, so on and so forth).

Additional features that can be added such as the ability to flip the image:

- Horizontally (about a vertical axis)
- Vertically (about a horizontal axis)
- Diagonal flip (flip along one axis first, then along the other)

These features are added in addition to the regular copy operation and need to be implemented directly within the copy function itself, such that the copy and flip happen simultaneously, not copy first then flip.

Horizontal flip:

Here, the first column of the first super-pixel needs to be copied onto the last column of the image buffer and the same follows for subsequent columns. This can be done by changing the width offset to the last column of the image buffer such that the start element will point to the last element of a row. The first element of the super-pixel will then be copied onto the last element's address of the subsequent row we are trying to copy onto. As we keep copying a super-pixel, the offset will reduce by 10 (super-pixels are of size 10*10).

Vertical flip:

The logic here is similar to horizontal flip wherein the start location for copy will have to point to the last row's first element in the image buffer. But here, we will have to change the width offset and height offset parameters. The height offset will have to be changed once for every three changes in the width parameter.

Formulae used:

element_offset = no. of elements in a row * row offset + column offset

element = origin + element_offset

Dependencies:

The main dependency here is with regards to the memory being consumed. For the time being I am making use of statically declared arrays of size 5*5 representing each macropixel and one super-pixel will have a size of 10*10 since it is comprised of 4 macropixels. This can later be changed to dma based allocation. With regards to the main thread communicating with the individual macropixel threads, only one interaction can happen at a time; Thus, it's necessary that the resources of the main thread are locked as soon as it starts its operation with another thread.

Testing:

TBD