Kieran Fergus and Javi Gonzalez
COMP 40
HW 4 Design Plan: arith

Implementation plan:

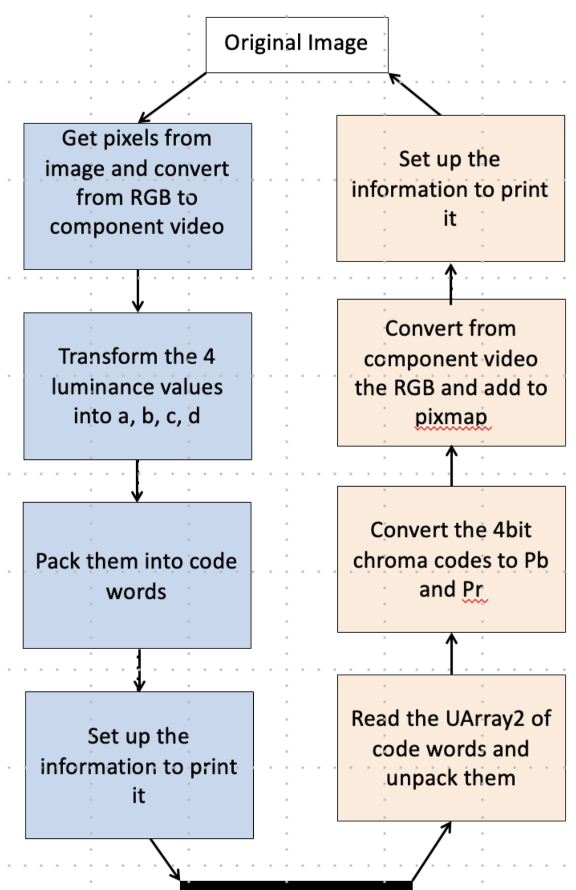
<u>Data structures we are going to use:</u>

- We are going to use **UArray2** to represent the **image** (we will populate every position on the original image with its respective in the new one)
  - This new array will have a block size of 2 to represent the 2 by 2 blocks we'll use
- We are going to use a **struct** to represent the **code words**
  - Containing uint64_t a, int64_t b, and so on…
- We will also create a **struct** to represent the **changed pixels**. Similar to Pnm_rgb, we will have Pnm_ypbpr that contains floats with the y, Pb, and Pr values. We'll set a pointer to it
- A **UArray** will also be used to store the **sequence of code words** we need to print in the end, here every element will correspond to a struct of code words

Organization:

We plan to have:

- .h and .c file for each step and respective counter-step of our implementation
  - The interfaces in these files will build upon each other so that we don't have too many separate interfaces to edit
- a unit testing .h and .c file



COMPRESSION

02/28

1. Read in the image using the pnm.h interface **(1 hour)**
    a. This will take and the *file pointer as a parameter* and will create a UArray2 of Pnm_rgb pixels with the pnm.h reading function, each containing different values of red, green and blue and we will
    b. Get the dimensions of the pixmap and if they are odd, trim them. *(we will pass in the UArray2)*
        i. Create a new UArray2b with the even dimensions and blocksize 2
        ii. We will also set the mapping default as block major mapping
        iii. Create an apply function copy pixel to populate the new image
    c. Delete the old map and *return the new UArray2b*

02/29

2. Convert UArray2b of Pnm_rgb pixels to floats **(3 hours)**
    a. Traverse the Uarray2 and access pixels one by one (mapping function so those parameters: col, row… - probably we will also have a struct containing the UArray2_Methods and other information we might need throughout the program)
    b. Create apply function to cast each red, blue, and green member value to type float
    c. Once we have each of this values of a pixel use the arithmetic given by the spec to get the Y, Pb and Pr values:
        i. y = 0.299 * r + 0.587 * g + 0.114 * b;
        pb = -0.168736 * r - 0.331264 * g + 0.5 * b;
        pr = 0.5 * r - 0.418688 * g - 0.081312 * b;
        ii. Store these values in a previously created struct called Pnm_ypbpr that contains the y, pb and pr values as floats
            1. Don't forget to check for the ranges of these values before returning the struct
        iii. Return the struct Pnm_ypbpr

03/01

3. Take UArray2b of Pnm_ypbpr structs and find the coefficients Pb average, Pr average, a, b, c, and, d **(6 hours)**
    a. Use the discrete cosine function to convert the 4 luminance values into the coefficients a,b,c and d
    b. Calculate the averages for Pb and Pr by dividing the Pb and Pr values in the coefficient struct by 4 (information is lost here as we are computing the average)
        i. Create a new mapping function that only visits the top left cell of each block
            1. This can be something like:

a. **int row = 0;**
for (int i = 0; i < total number of pixels / 4; i +=2) {
if (i == width) i = 0;
row++;
ACCESS EACH ELEMENT IN THE BLOCK
(i, row),(i + 1, row),(i, row + 1), (i + 1, row + 1)
AND ADD TO A LOCAL VAR TO GET AVG

c. Quantize (information is lost here as we round up values) b, c and d into 5 bit signed values, and using bitpack.h pack all of the information in a struct called code_word containing uint64_t a, int64_t b, int64_t c, int64_t d, **uint64_t pb and uint64_t pr**
   i. These last values will correspond to the index of the average we calculated with the Arith40_index_of_chroma function
d. *Return the struct with the new code word.*

03/02

4. Bit packing – putting the values of the struct into a code word **(5 hours)**
   a. Implement bitpack.c
   b. Use bitpack_fits() to determine if each struct value fits or not into the appropriate number of bits as defined by the table on spec page 5
   c. Get the field information we need using bitpack_get() and update them using bitpack_new()

03/02

5. We will call the previous function with a mapping function that will populate the Uarray of code words **(3 hours)**
   a. Ensure that we correct the little Endian order to Big Endian
   b. We can do this as we need to store these values in row-major order so in a UArray the order would already be the one wanted

03/03

6. Once we have the UArray populated we should create a function that prints the information out **(1 hour)**
   a. printf("COMP40 Compressed image format 2\n%u %u\n", width, height);
   b. Iterate through the map and print out the code words UArray as the compressed image using putchar

TESTING

Compression:

1. Reading in image to UArray2b

    a. (see cases below)
2. Converting UArray2b of Pnm_rgb pixels to floats
    a. Unit test/diff test that each RGB value is unchanged in float form after casting
    b. Use unit tests to assert that the formulas were correct in going from Pnm_rgb to Pnm_ypbpr
3. Going from UArray2b of Pnm_ypbpr structs and find the coefficients Pb average, Pr average, a, b, c, and, d
    a. Use unit test to assert that the mapping function(s) copy a,b,c,d, Pb_avg, and Pr_avg from each 2x2 block into the coefficient structs in the new UArray
4. Bit packing
    a. Unit test our implementation of bitpack to ensure that the values of each coefficient struct can be packed into a codeword correctly
5. Populating the UArray of codewords
    a. Unit test to ensure that each codeword is converted from Little endian order to Big endian order
    b. Unit test to ensure that the codewords are in order
6. Printing the compressed image
    a. Diff test the output of our compression algorithm with compressed images

DECOMPRESSION

<u>02/28</u>

1. Read the header of the compressed file **(2 hours)**
    a. To read the information use:
    unsigned height, width;
    int read = fscanf(in, "COMP40 Compressed image format 2\n%u %u", &width, &height);
    assert(read == 2);
    int c = getc(in);
    assert(c == '\n');

<u>02/28</u>

2. Allocate a new UArray2b of blocksize 2 with those dimensions **(30 mins)**
    a. Populate the information of the pixmap
    struct Pnm_ppm pixmap = { .width = width, .height = height , .denominator = denominator, .pixels = array , .methods = methods };
    b. Check for the denominator's constraints

<u>02/29</u>

3. Read the sequence of code words (UArray of structs) and check for the required runtime errors **(30 mins)**

<u>03/02</u>

4. Use bitpack_getu to unpack the values for a,b,c and d, as well as the index of Pb and Pr **(3 hours)**
    a. Use Artih40_index_of_chroma to get the values of Pb and Pr
    b. Use the inverse discrete cosine function to compute the values of y1, y2, y3 and y4 from a,b,c and d.
        i. **y1 = a − b − c + d**
           y2 = a − b + c − d
           y3 = a + b − c − d
           y4 = a + b + c + d

<u>02/29</u>

5. Now that we have the component video representation of the pixels (hence the values of y, pb and pr), convert back to RGB **(3 hours)**
    a. **r = 1.0 * y + 0.0 * pb + 1.402 * pr;**
       g = 1.0 * y - 0.344136 * pb - 0.714136 * pr;
       b = 1.0 * y + 1.772 * pb + 0.0 * pr;
    b. Quantize the values of r,g and b to fit in a suitable range (according to the denominator - some information is lost here due to rounding up values)

<u>03/01</u>

6. Put the RGB values in the pixel into pixmap->pixels to avoid repeated quantization **(2 hours)**
7. Once we have the pixmap->pixels populated call the Pnm ppmwrite(stdout, pixmap) to print the image to standard output **(30 mins)**

TESTING

<u>Decompression:</u>

1. Check that the order of the code words is Big Endian
2. Check for the correct creation of the 2D array
3. Check for correct population of dimensions and information about the 2D array
4. Check for general flow of program

Test cases:

- Empty image
- Odd dimensions

- Even dimensions
- Wrong header (Pnm.h should deal with this)
- Information isn't ordered as it should
-

General flow of program

1. The main idea of testing throughout the program is to follow a "back and forth" order implementing the functions. Eg: RGB to component video -> check if component video is right -> implement the component video to RGB
2. Test the interface and implementation of bitpack.h/c thoroughly to ensure that they work as expected
    a. One of the main cases is shifting the fields by 64

Test cases:

- Different combinations of wrong command lines