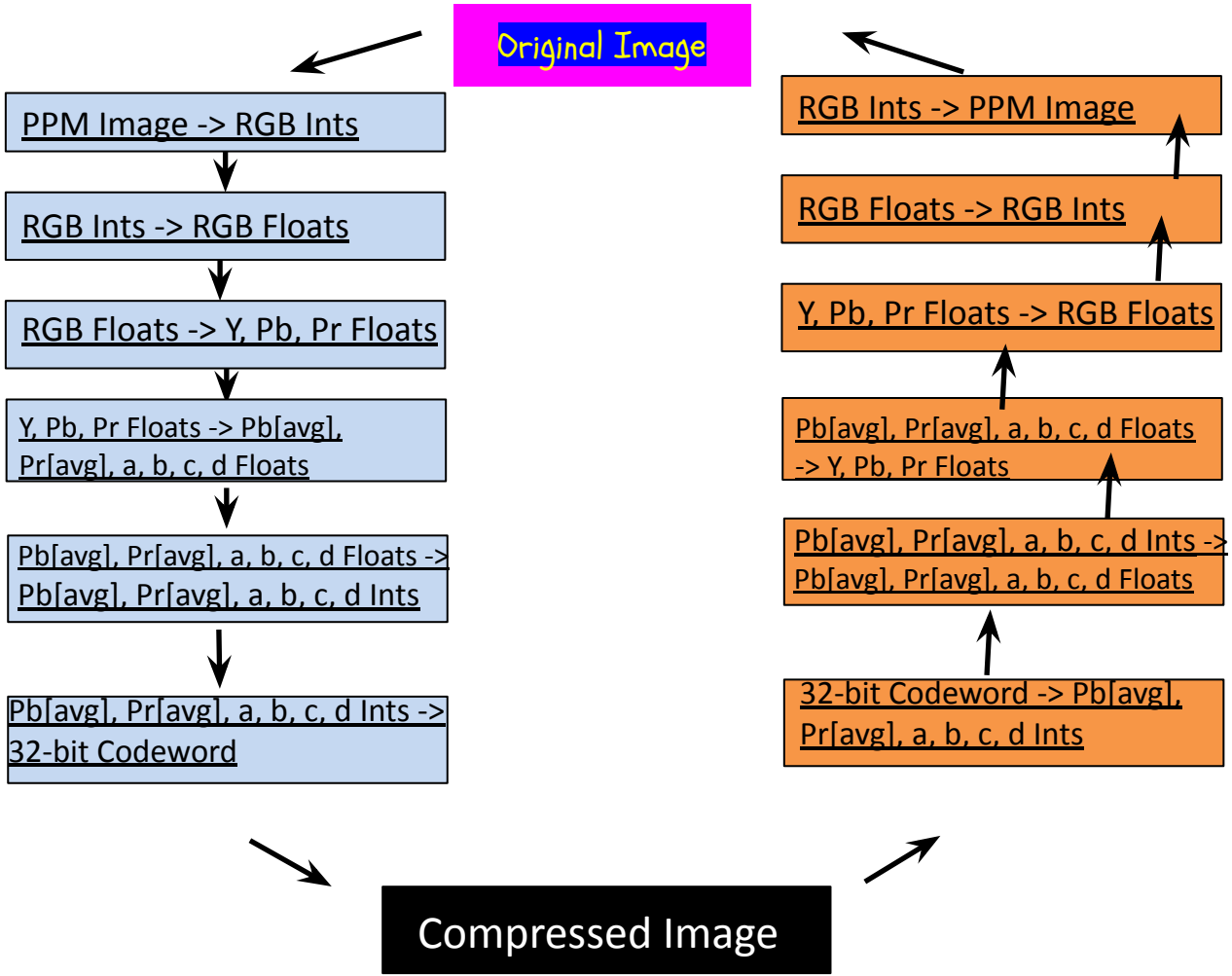
Arith Design Doc

Adam Weiss and Auriel Wish


Original Image
PPM Image -> RGB Ints
RGB Ints -> RGB Floats
RGB Floats -> Y, Pb, Pr Floats
Y, Pb, Pr Floats -> Pb[avg], Pr[avg], a, b, c, d Floats
Pb[avg], Pr[avg], a, b, c, d Floats -> Pb[avg], Pr[avg], a, b, c, d Ints
Pb[avg], Pr[avg], a, b, c, d Ints -> 32-bit Codeword
Compressed Image
32-bit Codeword -> Pb[avg], Pr[avg], a, b, c, d Ints
Pb[avg], Pr[avg], a, b, c, d Ints -> Pb[avg], Pr[avg], a, b, c, d Floats
Pb[avg], Pr[avg], a, b, c, d Floats -> Y, Pb, Pr Floats
Y, Pb, Pr Floats -> RGB Floats
RGB Floats -> RGB Ints
RGB Ints -> PPM Image

A note about testing

The basis of our testing for the compression steps is to manually do the calculations and compare the results with the program's results (which we will print out). Once we are sure any given compression step is working properly, we can test its corresponding decompression step simply by passing in the output of the compression step and ensuring that the decompression output is the same as the input to the compression.

We will test using the provided image files as well as our own custom ones which will include: 1xH and Wx1 images (which should compress into nothing), a 5x5 image whose output we can manually predict at every stage

PPM -> RGB Ints

- Use the pnm.h read function to read in PPM and output Pnm_ppm which contains an A2Methods_UArray2 that stores structs with the RGB integer values of the pixels
- No information is lost here
- Testing:
  - Use the pnm.h write function to output the image
  - Diff test this output with the original image

RGB Ints -> PPM

- Use the pnm.h write function to write the output Pnm_ppm to standard output
- The input is a Pnm_ppm with the A2methods_UArray2 of structs containing RGB ints
- No information is lost here
- Testing:
  - Use the corresponding compression stage output as the input for this stage and make sure that this stage’s output is the same as the compression stage’s input

RGB Ints -> RGB Floats

- Make new UArray2 called rbg_floats to store structs containing the RGB values as floats
- Divide each RGB int by the denominator value and place it into rgb_floats at corresponding location.
- The input here will be the A2methods_UArray2 in the pnm_ppm and the output will be the UArray2 of float pixel rgb values
- No information is lost here
- Testing
    - Print an RGB int UArray2 to an output file (that we ourselves can read). Read that file into program, put data into a UArray2, and send it through this compression stage. Print out the resulting UArray2. Since we have the input file, we know what the output should be (by manually doing the calculations). Compare the results.

RGB Floats -> RGB Ints

- Make a new UArray2 whose struct's values will be our chosen denominator (probably 255) multiplied by the corresponding float values
- The input is a UArray2 of structs containing RGB floats, the output is a Pnm_ppm containing an A2methods_UArray2 of structs containing RGB ints
- No information lost here (unless there is slight rounding when going from floats to ints, but that should be minimal if we choose a good denominator)
- Testing:
  - Use the corresponding compression stage output as the input for this stage and make sure that this stage's output is the same as the compression stage's input

RGB Floats -> Y, P_b, P_r Floats

- Convert each pixel from rgb space to component video space using the provided equations in the spec.
- The input at this step will be the 2D array of float pixel rgb values from the previous step and the output will be a 2D array of float pixel component video values. We will create a struct to hold the three component video pixel values and the output 2D array will contain these structs.
- No data is lost here
- Testing:
  - Same type of testing as before - print the RGB floats UArray2 to an output file, send that into this compression stage, print out the resulting UArray2, and compare those values with our manually calculated ones

Y, P_b, P_r Floats -> RGB Floats

- Convert the data in structs in a Y, Pb, Pr Floats UArray2 to RGB floats using the equations supplied in the spec
- The input is the Y, Pb, Pr Floats UArray2 and the output is the RGB floats UArray2
- No information is lost here
- Testing:
  - Use the corresponding compression stage output as the input for this stage and make sure that this stage's output is the same as the compression stage's input

Y, P_b, P_r Floats -> $P_{b[avg]}$, $P_{r[avg]}$, a, b, c, d Floats

- Convert the component video Y value into a, b, c, and d values using discrete cosine transformation.
- The input to this step will be the 2D array of component video value structs from the last step and the output will be a new 2D array of structs containing the $P_{b[avg]}$, $P_{r[avg]}$, a, b, c, and d values. This new array will be ¼ of the size as the input because these new values are an average of a 2x2 block of pixels from the input array.
- Data is lost here when converting the Y, P_b, and P_r because the 4 pixels in each 2x2 block are averaged
- Testing:
  - Once again, print the inputted and outputted UArray2s of this stage and compare the output UArray2 values with our manual calculations performed on the input UArray2

$P_{b[avg]}$, $P_{r[avg]}$, a, b, c, d Floats -> Y, P_b, P_r Floats

- Using the inverse of the discrete cosine function, calculate the Y_i values
- There is no way to undo the average of the P_b and P_r values, so those don't change
- The input is a UArray2 of structs containing P_b[avg], P_r[avg], a, b, c, d Floats, and the output is a UArray2 of structs containing Y, P_b, P_r Floats
- No information is lost here - the information has already been lost in the corresponding compression stage
- Testing
  - Use the corresponding compression stage output as the input for this stage and make sure that this stage's output is the same as the compression stage's input

$P_{b[avg]}$, $P_{r[avg]}$, a, b, c, d Floats -> $P_{b[avg]}$, $P_{r[avg]}$, a, b, c, d Ints

- Use Arith40_index_of_chroma to convert the $P_{b[avg]}$ and $P_{r[avg]}$ float values to integers.
- To convert the a, b, c, d floats to ints, multiply each value by 50 and round.
  - This will place them at integers between [-15, 15]
  - Note - if the values are < -0.3 or > 0.3, they should be automatically mapped to -15 and 15 respectively
- The input will be a UArray2 of all the values in floats, and the output will be a UArray2 of all the values in ints
- Information is lost here - quantization causes information loss because it sends a given range of values to one single value
- Testing:
  - From a given inputted UArray2, perform the quantization manually and also send it through this compression stage. Print out the program results and compare our results with our program's results

P_b, P_r, a, b, c, d Ints -> $P_{b[avg]}$, $P_{r[avg]}$, a, b, c, d Floats

- Use Arith40_chroma_of_index to get the P_b and P_r float values from their int values and convert a, b, c, d to their float forms
- The input is a UArray2 of structs containing P_b, P_r, a, b, c, d Ints, and the output is a UArray2 of structs containing P_b, P_r, a, b, c, d floats
- No information is lost here - the information has already been lost in the corresponding compression stage
- Testing:
  - Use the corresponding compression stage output as the input for this stage and make sure that this stage's output is the same as the compression stage's input

P_b, P_r, a, b, c, d Ints -> 32 bit codeword

- Use our bitpacking interface to pack these values into a 32 bit codeword
- Testing:
  - The testing for this stage will mostly be done when testing our bitpacking interface
  - However, we will still print out the codewords and make sure they align with what we expect them to be

32 bit codeword -> P_b, P_r, a, b, c, d Ints

- Unpack the codeword into the corresponding averaged component video values and DCT.
- Testing:
  - Use the corresponding compression stage output as the input for this stage and make sure that this stage's output is the same as the compression stage's input

Implementation Plan