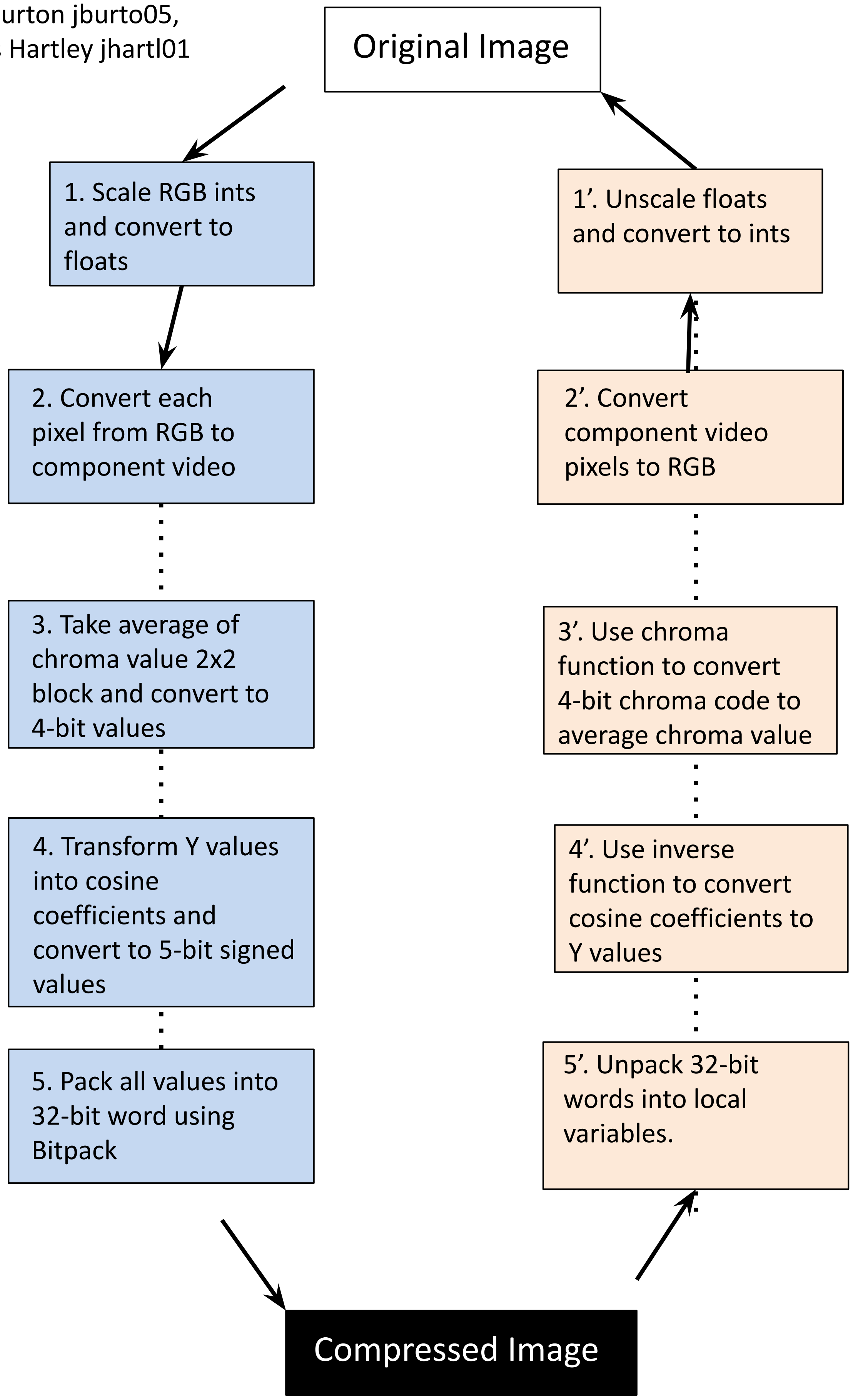
Jack Burton jburto05,
James Hartley jhartl01


Original Image
1. Scale RGB ints and convert to floats
2. Convert each pixel from RGB to component video
3. Take average of chroma value 2x2 block and convert to 4-bit values
4. Transform Y values into cosine coefficients and convert to 5-bit signed values
5. Pack all values into 32-bit word using Bitpack
Compressed Image
5’. Unpack 32-bit words into local variables.
4’. Use inverse function to convert cosine coefficients to Y values
3’. Use chroma function to convert 4-bit chroma code to average chroma value
2’. Convert component video pixels to RGB
1’. Unscale floats and convert to ints

Steps 1 and 1’

1. Scale RGB ints and convert to floats
   a. Scale each red, green, blue value for each pixel in PPM by the ppm denominator
   b. Initialize floats red, green, blue to those scaled values

1’. Unscale red, green, blue floats and convert to ints
   a. Unscale each red, green, blue float value for each pixel in PPM using ppm denominator
   b. Initialize ints red, green, blue to those unscaled values

STEP 1/1’ TESTING:

1. Provide invalid ppms, ensure that pnm or some part of program catches the invalid input
2. Scale valid ppms and ensure that the number of output pixels is equal to the number of input pixels
3. Provide various valid ppms, scale them, and pipe that to an output file. Give this output file to the unscaling function, and ensure that the result is the original ppm unmodified (can be checked using ppmdiff)
4. Run with valgrind to ensure no memory leaks

Inputs:

1) Pnm_rgb struct

1’) float red, float green, float blue (scaled values)

Outputs:

1) float red, float green, float blue (scaled values)

1’) int red, int green, int blue

Information is (potentially) lost on this step since we trim one row/column off of the Ppm if it has an odd height or width.

Steps 2 and 2’

2. Take each pixel and convert it to component video from RGB
    a. Take the red and blue values, and use the given spec equations to convert values to component video
    b. Store these y, Pb, and Pr values for each pixel

2’. Take each component video pixel and convert back to RGB
    a. Use the decompression spec equations to convert the y, Pb, and Pr values back to red, green, and blue pixel values
    b. Store these red, green, blue values for each pixel

STEP 2/2’ TESTING:

1. Print y, Pb, and Pr values from various RGB values and ensure the numbers are in reasonable component video ranges (based on paragraph in spec)
2. Take an RGB pixel, convert it to component video, and put this into the function that converts back to RGB. Ensure the RGB values match.
3. Run with a full raster of pixels (a ppm), and run with valgrind. Ensure no memory leaks.

Inputs:
2) Float red, float green, float blue (scaled pixel float values)
2’) float y, float Pb, float Pr *(potentially stored in a new struct Pnm_video)*
Outputs:
2) float y, float Pb, float Pr (component video values) *(potentially from struct Pnm_video)*
2’) float red, float green, float blue

Information is not lost on this step.

Steps 3 and 3’

3. Create a block major mapping that maps in 2x2 blocks over the ppm, and for each block take the averages of the Pb and Pr values–then convert these to 4-bit values using given Arith40_index_of_chroma and store them.

3’. Convert 4-bit chroma codes to Pb and Pr averages using the given Arith40_chroma_of_index, and store these.

STEP 3/3’ TESTING:

1. Implement UArray2b with blocksize 2, and map over it, ensuring that it accesses elements as expected.
2. Build a function to take averages of Pb and Pr, and ensure it returns correct values for sample Pr/Pb values
3. Run the given function to convert those averages to 4-bit chroma codes–try running the given inverse function and ensure it returns the correct average Pb/Pr values.
4. Valgrind, test compression/decompression on an actual ppm and ensure no memory leaks

Inputs
3) float Pb, float Pr
3’) 4-bit chroma code (i.e. index for internal table from given function)
Outputs
4) 4-bit chroma code (i.e. index for internal table)
4’) float PbAvg, float PrAvg

Information IS lost on this step because there is no way to “un-average” the Pb and Pr values for a 2x2 block–this average provides an approximate representation of these 4 pixels, but you cannot get the exact 4 pixel values back.

Steps 4 and 4’

4. Transform Y values into cosine coefficients and convert to 5-bit signed values
    a. Use a discrete cosine transform equation to transform 4 Y values into coefficients a, b, c, d
    b. Convert b, c, d, values into 5-bit signed values

4’. Use inverse function to convert cosine coefficients to Y values
    a. Compute Y_1,Y_2,Y_3, and Y_4 from a, b, c, and d

STEP 4/4’ TESTING:

1. Do sizeof() on output values from cosine transformation to ensure a is 9 bits and b/c/d are 5 bits
2. Test cosine coefficients that are close and ensure they convert to the same standard value on output (between -0.3 and +0.3)
3. Put output values into inverse function and ensure you get original Y1-4 values back
4. Run on sample ppm with valgrind, ensuring it works and no memory leaks

Inputs:
2) unsigned ints Y_1,Y_2,Y_3, and Y_4
2’) cosine coefficients a (9-bit signed), b, c, and d (5-bit signed)
Outputs:
2) cosine coefficients a (9-bit signed), b, c, and d (5-bit signed)
2’) unsigned ints Y_1,Y_2,Y_3, and Y_4

Information IS lost on this step because you are converting a range of different values into a single value, and there is no way to reverse it and get the original value.

Steps 5 and 5’

5. Pack all values into 32-bit word using Bitpack
    a. Following big-endian order, put components a, b, c, d, Pb index and Pr index into the code word in spec-specified order using putchar()

5’. Unpack all values into 32-bit word using Bitpack
    a. Read in 32-bit code word in sequence and throw CRE if not enough codewords/incomplete last codeword

STEP 5/5’ TESTING:

1. Unpack components of the 32-bit codeword, ensuring that each component has the correct LSB
2. Run Bitpack on various sets of components, ensure codeword size is 32 bits using sizeof()
3. Create components from sample PPM, pack them into 32-bit word, and unpack them. Ensure same component values are retrieved.
4. Run process of packing/unpacking with valgrind, ensuring no memory leaks

Inputs:
2) cosine coefficients a (9-bit signed), b, c, and d (5-bit signed), indexes of P_B and P_R
2’) 32-bit code words
Outputs:
2) 32-bit code words
2’) cosine coefficients a (9-bit signed), b, c, and d (5-bit signed) indexes of P_B and P_R

Information is not lost on this step.