* 1. An example of a non-transcendental number that would not be represented accurately with bfloat16 is +5.55E-130. This is because the number has an exponent that exceeds the bounds of the 8 bits used in bfloat16. (The number is too small to be represented)
  2. typedef short int int16\_t;

#define ITERS 16 // Resolution of Taylor Series approximation

int16\_t sin16(int16\_t x) {

int16\_t result = x;

// Compute the Taylor series expansion of sine

for (int16\_t i = 1; i < ITERS; i++) {

int16\_t termVal = (2 \* i) + 1; // Odd numbers for 3, 5, 7, 9, ...

int16\_t numerator = 1; int16\_t denom = 1;

// Compute numerator

for (int16\_t i = 0; i < termVal; i++)

numerator \*= x;

// Compute denominator

for (int16\_t i = termVal; i > 0; i--)

denom \*= i;

// Add to the result

if (i % 2 == 0) {

result += (numerator / denom);

} else {

result -= (numerator / denom);

}

}

return result;

}

int16\_t cos16(int16\_t x) {

int16\_t result = x;

// Compute the Taylor series expansion of sine

for (int16\_t i = 1; i < ITERS; i++) {

int16\_t termVal = (2 \* i); // Even numbers for 2, 4, 6, 8, ...

int16\_t numerator = 1; int16\_t denom = 1;

// Compute numerator

for (int16\_t i = 0; i < termVal; i++)

numerator \*= x;

// Compute denominator

for (int16\_t i = termVal; i > 0; i--)

denom \*= i;

// Add to the result

if (i % 2 == 0) {

result += (numerator / denom);

} else {

result -= (numerator / denom);

}

}

return result;

}

* 1. Assuming each processor has its own local cache (of layers L1 or more), it will need to update values of FFT\_input[i] anytime there is a cache miss due to the updating in Processor C. In the case of Processor B, this could prove much worse than in the other two due to reading from FFT\_input[i] two separate times during each iteration of the loop.
  2. First possible solution: have processor C complete its work before the work of processor A and B complete. In this scenario, A and B should not run into any cache misses during runtime.  
     Second possible solution: have the total work divided equally amongst the three processors (iterate through only N/3 elements of FFT\_input, copied to local cache). Each processor will also do the sum, max, and .real updating, but only to its own third of the array and only to its own local instance of the array. Then, when complete, each processor updates shared memory.

1. #include "lib/image.h" // Library from Data Structures and Algorithms class

#include <string>

#include <iostream>

// ===== Image functions ===================================

/\*\*

\* Function to load an image from a file and return as an Image object

\* @param filename - the name of the file to load

\* @return - the loaded image object as an Image<Pixel>

\*/

Image<Pixel> loadImage(const std::string &filename){

try {

// Try to load image given filename

return readFromFile(filename);

}

catch (std::exception &e){

// The file could not be loaded

std::cerr << "Error loading image: " << e.what() << std::endl;

Image<Pixel> emptyImage;

return emptyImage;

}

}

/\*\*

\* Writes an image to a file given the image and filename

\* @param image - the image to write

\* @param filename - the filename to write to

\*/

void writeImageToFile(const Image<Pixel> &image, const std::string &filename){

try {

writeToFile(image, filename);

}

catch (std::exception &e){

// The file could not be written

std::cerr << "Error writing image: " << e.what() << std::endl;

}

}

// ===== Image processing functions =========================

// Take in a pixel, and convert it to black and white

Pixel toBlackAndWhite(Pixel pixel){

// Get the average of the RGB values

uint8\_t average = (pixel.red + pixel.green + pixel.blue) / 3;

// Set the RGB values to the average

Pixel newPixel = {average, average, average, 255};

return newPixel;

}

// Take in an image, and convert all pixels to black and white

Image<Pixel> imageToBlackAndWhite(Image<Pixel> image){

// Create a new image with the same dimensions as the original

Image<Pixel> newImage(image.width(), image.height());

// Loop through all pixels in the image  
# pragma omp for

for (int i = 0; i < image.width(); i++){  
# pragma omp for

for (int j = 0; j < image.height(); j++){

// Get the pixel at the current position

// For some reason, the image library uses (y, x) instead???

Pixel pixel = image(j, i);

// Convert the pixel to black and white

Pixel newPixel = toBlackAndWhite(pixel);

// Set the pixel in the new image

newImage(j, i) = newPixel;

}  
# pragma omp barrier

}  
# pragma omp barrier

return newImage;

}

// Take in a pixel, and find the closest palette color to it

Pixel findClosestPaletteColor(Pixel pixel){

// Start by extracting the RGB values as an average

uint8\_t average = (pixel.red + pixel.green + pixel.blue) / 3;

// Compute a rounding of this average

uint8\_t newColor = 0;

if (average < 128)

newColor = 255;

// Return the pixel

Pixel newPixel = {newColor, newColor, newColor, 255};

return newPixel;

}

// Find the quantitative error of a new pixel compared to an old pixel

uint8\_t findQuantitativeError(Pixel oldPixel, Pixel newPixel){

// Get the average of the RGB values

uint8\_t oldAverage = (oldPixel.red + oldPixel.green + oldPixel.blue) / 3;

uint8\_t newAverage = (newPixel.red + newPixel.green + newPixel.blue) / 3;

// Compute the error

uint8\_t error = oldAverage - newAverage;

return error;

}

// Update a pixel based on a passed in error

Pixel updatePixel(Pixel pixel, uint8\_t error){

// Get the average value of the pixel

uint8\_t average = (pixel.red + pixel.green + pixel.blue) / 3;

// Compute the new average

uint8\_t newAverage = average + error;

// Return the new pixel

Pixel newPixel = {newAverage, newAverage, newAverage, 255};

return newPixel;

}

// Take in an image, and apply Floyd-Steinberg dithering to it

Image<Pixel> applyDitheringToImage(Image<Pixel> image){

Image<Pixel> newImage(image.width(), image.height());

// Copy the old image to the new image

#pragma omp for

for (int y = 0; y < image.height(); y++){

for (int x = 0; x < image.width(); x++)

newImage(y, x) = image(y, x);

}

#pragma omp barrier

// Now loop through and apply dithering

#pragma omp for

for (int y = 0; y < newImage.height(); y++){

for (int x = 0; x < newImage.width(); x++){

// Save the old pixel

Pixel oldPixel = newImage(y, x);

// Create a new pixel with the nearest palette color

Pixel newPixel = findClosestPaletteColor(oldPixel);

// Compute the error

uint8\_t quantError = findQuantitativeError(oldPixel, newPixel);

// Set the new pixel into the image

newImage(y, x) = newPixel;

// Propagate the error to the surrounding pixels (if they exist)

if (x + 1 < newImage.width())

newImage(y, x + 1) =

updatePixel(newImage(y, x + 1), quantError \* 7 / 16);

if (x - 1 >= 0 && y + 1 < newImage.height())

newImage(y + 1, x - 1) =

updatePixel(newImage(y + 1, x - 1), quantError \* 3 / 16);

if (y + 1 < newImage.height())

newImage(y + 1, x) =

updatePixel(newImage(y + 1, x), quantError \* 5 / 16);

if (x + 1 < newImage.width() && y + 1 < newImage.height())

newImage(y + 1, x + 1) =

updatePixel(newImage(y + 1, x + 1), quantError \* 1 / 16);

}

}

#pragma omp barrier

// Now we've (hopefully) dithered an image

return newImage;

}

// ===== Main code ===========================================================

int main(){

std::cout << "Convert an image to black and white!" << std::endl;

// Image name - Must be a PNG file

const std::string \*filename = new std::string("lena.png");

// Create the input filename

const std::string \*inFilepath = new std::string("in/" + \*filename);

// Load the image

Image<Pixel> image = loadImage(\*inFilepath);

// Check if the image was loaded

if (image.width() == 0 || image.height() == 0){

std::cerr << "Could not load image!" << std::endl;

return EXIT\_FAILURE;

}

else{

std::cout << "Image successfully loaded. (" << image.width() << "x"

<< image.height() << ")" << std::endl;

}

// Convert the image to black and white

Image<Pixel> imageBW = imageToBlackAndWhite(image);

std::cout << "Image converted to black and white." << std::endl;

// Dither the image

Image<Pixel> imageDither = applyDitheringToImage(imageBW);

std::cout << "Image dithered." << std::endl;

// Create the output filename

const std::string \*outFilepath = new std::string("out/" + \*filename);

// Write the image to a file

writeImageToFile(imageDither, \*outFilepath);

std::cout << "Image successfully written to file." << std::endl;

return EXIT\_SUCCESS;

}

1. // A rough implementation. Hopefully this is the right idea. Wasn’t able to thoroughly test it  
   std::vector<float> computeArtificialReverb(){

// Signal stored in std::vector<float> anechoicRecording;

// Impulse stored in std::vector<float> impulseRecording;

// This is where we store windows

std::vector<std::vector<complex\_t>> fftAnechoicWindows = new std::vector<std::vector<complex\_t>>();

std::vector<std::vector<complex\_t>> fftImpulseWindows = new std::vector<std::vector<complex\_t>>();

std::vector<std::vector<complex\_t>> fftSums = new std::vector<std::vector<complex\_t>>();

// Split into windows

for (int i = 0; i < N; i += WINDOW\_SIZE){

// Save this window of the recording

std::vector<float> anechoicRecordingWindow = new std::vector<float>(2 \* WINDOW\_SIZE);

std::vector<float> impulseResponseWindow = new std::vector<float>(2 \* WINDOW\_SIZE);

for (int j = i; j < i + WINDOW\_SIZE; j++){

if (j < anechoicRecording.size()){

anechoicRecordingWindow(j - i) = anechoicRecording(j);

} else {

anechoicRecordingWindow(j - i) = 0;

}

if (j < impulseResponseWindow){

impulseResponseWindow(j - i) = impulseResponse(j);

} else {

impulseResponseWindow(j - i) = 0;

}

}

// Now pad with 1024 zeroes

for (int j = 1024; j < (2 \* WINDOW\_SIZE); j++){

anechoicRecordingWindow(j) = 0;

impulseResponseWindow(j) = 0;

}

// Now that we have this, FFT both

std::vector<complex\_t> fftAnechoic = fft2048(anechoicRecordingWindow);

fftAnechoicWindows.insert(fftAnechoic);

std::vector<complex\_t> fftImpulse = fft2048(impulseResponseWindow);

fftImpulseWindows.insert(fftImpulse);

fftSums.insert(new std::vector<complex\_t>(WINDOW\_SIZE \* 2));

for (int j = 0; j < fftAnechoicWindows.size(); j++){

for (int k = fftImpulseWindows.size(); k >= 0; k--){

std::vector<complex\_t> summedWindow = new std::vector<complex\_t>();

for (int l = 0; l < (2 \* WINDOW\_SIZE); l++){

// Compute the sum of two complex numbers here

complex\_t value = {

fftImpulseWindows.at(i).at(l).real + fftAnechoicWindows.at(i).at(l),

fftImpulseWindows.at(i).at(l).imag + fftAnechoicWindows.at(i).at(l)};

summedWindow.insert(value);

}

// Now insert into the sum

fftSums.insert(summedWindow);

}

}

}

// Now that we have the windows, compute the IFFT of each to get the time domain

std::vector<std::vector<float>> ifftSignals = new std::vector<std::vector<float>>();

for (int i = 0; i < fftSums.size(); i++){

ifftSignals.insert(ifft2048(fftSums.at(i)));

}

// Then, simply concatenate these time domain signals together

std::vector<float> timeDomainSignal = new std::vector<float>();

for (int i = 0; i < ifftSignals.size(); i++) {

for (int j = 0; j < ifftSignals.at(0).size(); j++) {

timeDomainSignal.insert(ifftSignals.at(i).at(j));

}

}

return timeDomainSignal;

}