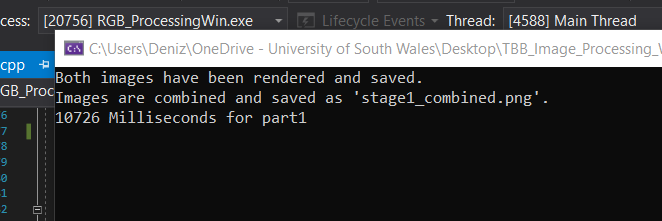
Parallel Programming Report

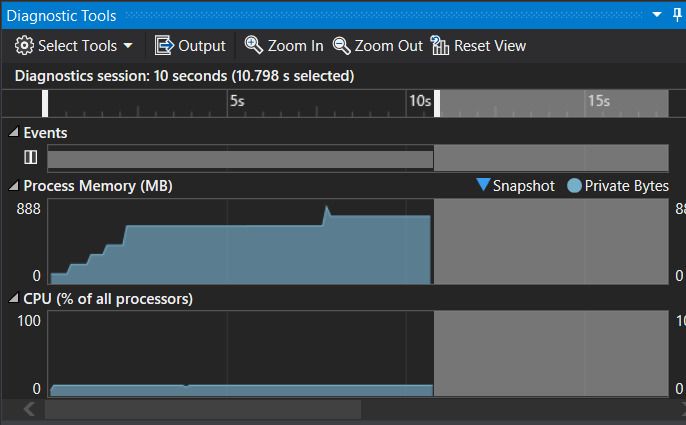
I ran my programme 5 times and calculated the averages and placed them in a chart above. I have placed all my results form the 5 tests bellow in a table.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Part1 | Part1multi | Part2 | Part2TBB | Part3 | Part3TBB |
| Run1 | 10930 | 9276 | 6383 | 8175 | 15716 | 15105 |
| Run2 | 10726 | 9797 | 6501 | 8230 | 15676 | 14635 |
| Run3 | 10951 | 9197 | 6727 | 8266 | 16047 | 15634 |
| Run4 | 10823 | 8990 | 6384 | 8043 | 15629 | 14941 |
| Run5 | 10968 | 9591 | 6483 | 7918 | 15738 | 15134 |
| AVERAGE | 10879.6 | 9370.2 | 6495.6 | 8126.4 | 15761.2 | 15089.8 |

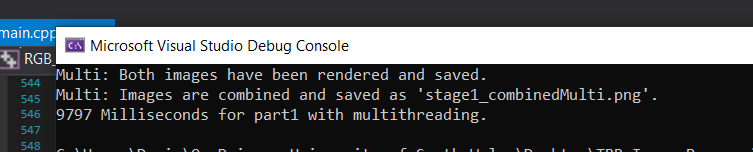
Part1

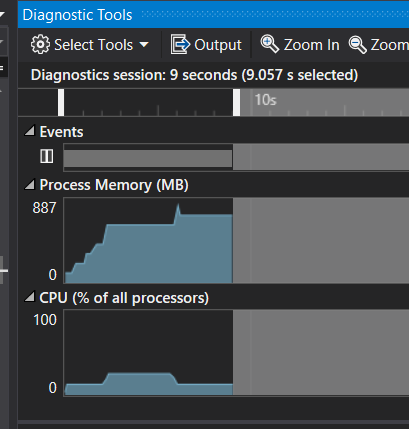
Part 1 Without Multi-Threading:





Part 1 With Multi-Threading:





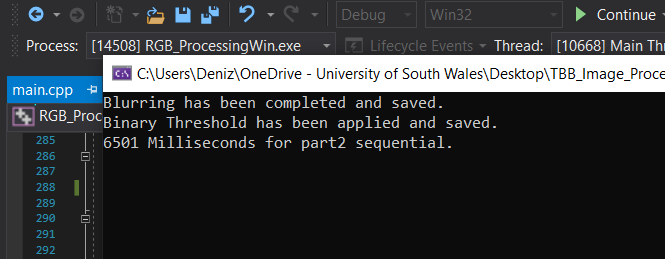
Part 1 Summary:

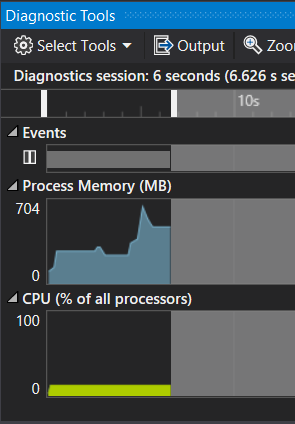
Without multi-threading part1 took 10726 milliseconds to complete the image comparison and rendering of the top and bottom and joining them together to make the complete output image. However, for the multithreading part1 it was noticeably quicker with 9796 milliseconds of runtime.

The process memory used isn’t much different with it only being 1 mb less than the non-multithreaded version but there is a significant change in CPU usage. The multithreaded version peaks at 25% for a shorter time whereas the non-multithreaded function is at a stable 13% usage for a longer duration.

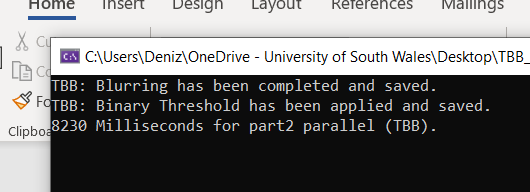
Part2

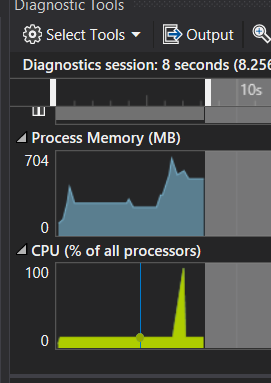
Part 2 Sequential



13%

Part 2 Parallel





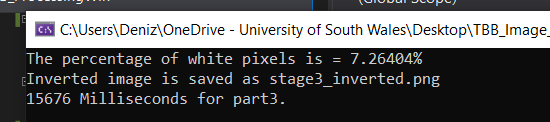
Part 2 Summary

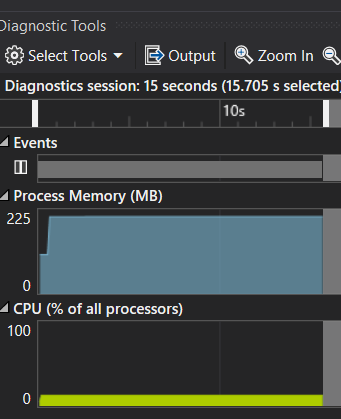
My sequential part 2 took 6501 milliseconds to apply the Guassian blur to the image to then apply the binary threshold to the image. However, my parallel part2 took 8230 milliseconds to complete these processes. I didn’t believe that this was true, so I tested my functions without the cout methods in it and calculated the time in the main and the results had changed to 8100 milliseconds. I couldn’t figure out why this was happening.

The CPU usage is what really stands out, in the sequential part 2 it stays at a stable 13% the whole time. How ever the CPU usage for the parallel function stays at 13% till it shoots up to above 90% and then back down to 13%.

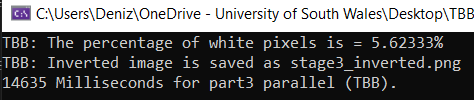
Part 3

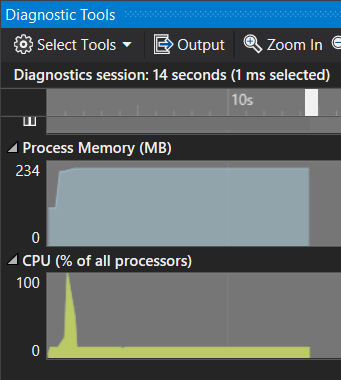
Part 3 Sequential





Part 3 Parallel





Parallel 3 Summary

Part 3 Sequential took 15676 to calculate the percentage of white pixels and inverting the image. The parallel part 3 took a noticeably less time of 14635 milliseconds to process everything.

The CPU usage is again something that is significantly different. The sequential function just like the other parts was stable at 13% whereas the parallel spiked to 100% of CPU usage, then dying back down to 13% steadily till all processes were completed.

APPENIX

#include <iostream>

#include <vector>

//Thread building blocks library

#include <tbb/task\_scheduler\_init.h>

#include <tbb/blocked\_range2d.h>

#include <tbb/parallel\_for.h>

//Free Image library

#include <FreeImagePlus.h>

#include <thread>

#include <chrono>

#include <mutex>

using namespace std;

using namespace tbb;

using namespace std::chrono;

void part1()

{

auto start = high\_resolution\_clock::now();

fipImage inputImageTop1, inputImageTop2, inputImageBottom1, inputImageBottom2;//creating the images

inputImageTop1.load("../Images/render\_top\_1.png");//loading the images

inputImageTop2.load("../Images/render\_top\_2.png");

inputImageBottom1.load("../Images/render\_bottom\_1.png");

inputImageBottom2.load("../Images/render\_bottom\_2.png");

auto width = inputImageTop1.getWidth();

auto height = inputImageTop1.getHeight();

fipImage renderTopImage = fipImage(FIT\_BITMAP, width, height, inputImageTop1.getBitsPerPixel());//setting the width height anf bits for the image created

fipImage renderBottomImage = fipImage(FIT\_BITMAP, width, height, inputImageBottom1.getBitsPerPixel());

RGBQUAD\* colours = new RGBQUAD();//creating the colours for processing

RGBQUAD\* first = new RGBQUAD();

RGBQUAD\* second = new RGBQUAD();

BYTE\* inputImageTop1Pixels = inputImageTop1.accessPixels();//accessing the pixels

BYTE\* inputImageTop2Pixels = inputImageTop2.accessPixels();

BYTE\* inputImageBottom1Pixels = inputImageBottom1.accessPixels();

BYTE\* inputImageBottom2Pixels = inputImageBottom2.accessPixels();

for (int y = 0; y < height; y++)//for loop for the height

{

for (int x = 0; x < width; x++)//for loop to go through the width

{

inputImageTop1.getPixelColor(x, y, first);//getting the pixel colour and seeting it to the variables

inputImageTop2.getPixelColor(x, y, second);

if (first->rgbBlue == second->rgbBlue && first->rgbGreen == second->rgbGreen && first->rgbRed == second->rgbRed)//if to check if they are equal to each other

{

colours->rgbBlue = 0;//setting the colours to black

colours->rgbGreen = 0;

colours->rgbRed = 0;

}

else

{

colours->rgbBlue = 255;//setting the colours to white

colours->rgbGreen = 255;

colours->rgbRed = 255;

}

renderTopImage.setPixelColor(x, y, colours);//setting the colours

inputImageBottom1.getPixelColor(x, y, first);//getting the pixel colour for each half

inputImageBottom2.getPixelColor(x, y, second);

if (first->rgbBlue == second->rgbBlue && first->rgbGreen == second->rgbGreen && first->rgbRed == second->rgbRed)

{

colours->rgbBlue = 0;

colours->rgbGreen = 0;

colours->rgbRed = 0;

}

else

{

colours->rgbBlue = 255;

colours->rgbGreen = 255;

colours->rgbRed = 255;

}

renderBottomImage.setPixelColor(x, y, colours);

}

inputImageTop1Pixels += inputImageTop1.getScanWidth();//scanning the width and adding it to my input image

inputImageTop2Pixels += inputImageTop2.getScanWidth();

inputImageBottom1Pixels += inputImageBottom1.getScanWidth();

inputImageBottom2Pixels += inputImageBottom2.getScanWidth();

}

renderTopImage.save("../Images/stage1\_top.png");//saving results

renderBottomImage.save("../Images/stage1\_bottom.png");

cout << "Both images have been rendered and saved." << endl;

BYTE\* renderTopImagePixels = renderTopImage.accessPixels();//accessing the pixels for rendering

BYTE\* renderBottomImagePixels = renderBottomImage.accessPixels();

fipImage composite;

composite = fipImage(FIT\_BITMAP, width, height, inputImageBottom1.getBitsPerPixel());

for (int y = 0; y < height; y++)

{

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

{

renderTopImage.getPixelColor(x, y, first);

renderBottomImage.getPixelColor(x, y, second);

colours->rgbBlue = (first->rgbBlue + second->rgbBlue) \* 0.5;//joining the colours of both images into one

colours->rgbGreen = (first->rgbGreen + second->rgbGreen) \* 0.5;

colours->rgbRed = (first->rgbRed + second->rgbRed) \* 0.5;

composite.setPixelColor(x, y, colours);

}

renderTopImagePixels += renderTopImage.getScanWidth();

renderBottomImagePixels += renderBottomImage.getScanWidth();

}

composite.save("../Images/stage1\_combined.png");

cout << "Images are combined and saved as 'stage1\_combined.png'. " << endl;

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << duration.count() << " Milliseconds for part1" << endl;

}

void part1multi()

{

auto start = high\_resolution\_clock::now();

fipImage inputImageTop1, inputImageTop2, inputImageBottom1, inputImageBottom2;

inputImageTop1.load("../Images/render\_top\_1.png");

inputImageTop2.load("../Images/render\_top\_2.png");

inputImageBottom1.load("../Images/render\_bottom\_1.png");

inputImageBottom2.load("../Images/render\_bottom\_2.png");

auto width = inputImageTop1.getWidth();

auto height = inputImageTop1.getHeight();

fipImage renderTopImage = fipImage(FIT\_BITMAP, width, height, inputImageTop1.getBitsPerPixel());

fipImage renderBottomImage = fipImage(FIT\_BITMAP, width, height, inputImageBottom1.getBitsPerPixel());

RGBQUAD\* colours = new RGBQUAD();

RGBQUAD\* colours1 = new RGBQUAD();

RGBQUAD\* first = new RGBQUAD();

RGBQUAD\* first1 = new RGBQUAD;

RGBQUAD\* second = new RGBQUAD();

RGBQUAD\* second1 = new RGBQUAD();

BYTE\* inputImageTop1Pixels = inputImageTop1.accessPixels();

BYTE\* inputImageTop2Pixels = inputImageTop2.accessPixels();

BYTE\* inputImageBottom1Pixels = inputImageBottom1.accessPixels();

BYTE\* inputImageBottom2Pixels = inputImageBottom2.accessPixels();

vector<thread> threads;

threads.push\_back(thread([&]() {//pushng back the thread in the vector creating a thread for the first bit

for (int y = 0; y < height; y++)

{

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

{

inputImageTop1.getPixelColor(x, y, first);

inputImageTop2.getPixelColor(x, y, second);

if (first->rgbBlue == second->rgbBlue && first->rgbGreen == second->rgbGreen && first->rgbRed == second->rgbRed)

{

colours->rgbBlue = 0;

colours->rgbGreen = 0;

colours->rgbRed = 0;

}

else

{

colours->rgbBlue = 255;

colours->rgbGreen = 255;

colours->rgbRed = 255;

}

renderTopImage.setPixelColor(x, y, colours);

}

inputImageTop1Pixels += inputImageTop1.getScanWidth();

inputImageTop2Pixels += inputImageTop2.getScanWidth();

}

renderTopImage.save("../Images/stage1\_topMulti.png");

}));

threads.push\_back(thread([&]() {// pushing back the thread into the vector

for (int y = 0; y < height; y++)

{

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

{

inputImageBottom1.getPixelColor(x, y, first1);

inputImageBottom2.getPixelColor(x, y, second1);

if (first1->rgbBlue == second1->rgbBlue && first1->rgbGreen == second1->rgbGreen && first1->rgbRed == second1->rgbRed)

{

colours1->rgbBlue = 0;

colours1->rgbGreen = 0;

colours1->rgbRed = 0;

}

else

{

colours1->rgbBlue = 255;

colours1->rgbGreen = 255;

colours1->rgbRed = 255;

}

renderBottomImage.setPixelColor(x, y, colours1);

}

inputImageBottom1Pixels += inputImageBottom1.getScanWidth();

inputImageBottom2Pixels += inputImageBottom2.getScanWidth();

}

renderBottomImage.save("../Images/stage1\_bottomMulti.png");

cout << "Multi: Both images have been rendered and saved." << endl;

}));

for (thread& thread : threads)

{

thread.join();//joining the threads

}

fipImage composite;

composite = fipImage(FIT\_BITMAP, width, height, inputImageBottom1.getBitsPerPixel());

BYTE\* renderTopImagePixels = renderTopImage.accessPixels();

BYTE\* renderBottomImagePixels = renderBottomImage.accessPixels();

for (int y = 0; y < height; y++)

{

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

{

renderTopImage.getPixelColor(x, y, first);

renderBottomImage.getPixelColor(x, y, second);

colours->rgbBlue = (first->rgbBlue + second->rgbBlue) \* 0.5;

colours->rgbGreen = (first->rgbGreen + second->rgbGreen) \* 0.5;

colours->rgbRed = (first->rgbRed + second->rgbRed) \* 0.5;

composite.setPixelColor(x, y, colours);

}

renderTopImagePixels += renderTopImage.getScanWidth();

renderBottomImagePixels += renderBottomImage.getScanWidth();

}

composite.save("../Images/stage1\_combinedMulti.png");

cout << "Multi: Images are combined and saved as 'stage1\_combinedMulti.png'. " << endl;

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << duration.count() << " Milliseconds for part1 with multithreading." << endl;

}

void part2()

{

auto start = high\_resolution\_clock::now();

const int kHeight = 5;//setting the height width and sigma for the Guassian Kernal

const int kWidth = 5;

float sigma = 10.25f;

float kernelGaussian[kWidth][kHeight];

double sum = 0.0;

for (int i = 0; i < kWidth; i++)

{

for (int j = 0; j < kHeight; j++)

{

kernelGaussian[i][j] = exp(-(i \* i + j \* j) / (2 \* sigma \* sigma)) / (2 \* 3.142f \* sigma \* sigma);//guassian equation to create the kernal

sum += kernelGaussian[i][j];

}

}

for (int i = 0; i < kWidth; i++)//normalising the kernal

{

for (int j = 0; j < kHeight; j++)

{

kernelGaussian[i][j] /= sum;

}

}

fipImage inputImageStage1;//loading images and getting necesary values

inputImageStage1.load("../Images/stage1\_combined.png");

auto height = inputImageStage1.getHeight();

auto width = inputImageStage1.getWidth();

inputImageStage1.convertToFloat();

float\* inputBuffer = (float\*)inputImageStage1.accessPixels();

fipImage outputImageStage2Blurred;

outputImageStage2Blurred = fipImage(FIT\_FLOAT, width, height, 32);

float\* outputBuffer = (float\*)outputImageStage2Blurred.accessPixels();

const int border = (kWidth - 1) / 2;//creating the border for the image

for (int y = border; y < height - border; y++)//for loops to get all the pixels that are inside the border

{

for (int x = border; x < width - border; x++)

{

for (int i = -border; i < border; i++)//multiply the checked neighbouring pixels by kernal pixels

{

for (int j = -border; j < border; j++)

{

outputBuffer[y \* width + x] = outputBuffer[y \* width + x] + inputBuffer[(y - i) \* width + (x - j)] \* kernelGaussian[i + border][j + border];

}

}

}

}

outputImageStage2Blurred.convertToType(FREE\_IMAGE\_TYPE::FIT\_BITMAP);

outputImageStage2Blurred.convertTo24Bits();

outputImageStage2Blurred.save("../Images/stage2\_blurred.png");

cout << "Blurring has been completed and saved." << endl;

fipImage thresholdPixels;

thresholdPixels.load("../Images/stage2\_blurred.png");

auto tWidth = thresholdPixels.getWidth();

auto tHeight = thresholdPixels.getHeight();

thresholdPixels.convertToFloat();

float\* outputBuffer1 = (float\*)thresholdPixels.accessPixels();

fipImage blurredPixels;

blurredPixels = fipImage(FIT\_FLOAT, width, height, outputImageStage2Blurred.getBitsPerPixel());

blurredPixels.convertToFloat();

float\* threshHoldPixels = (float\*)blurredPixels.accessPixels();

float threshold = 0.0f;

float max = 255.0f;

for (int y = 0; y < tHeight; y++)

{

for (int x = 0; x < tWidth - 1; x++, outputBuffer1++, threshHoldPixels++)//for loops to run through the image

{

float currentPixel = outputBuffer1[x];

if (currentPixel > threshold)

{

threshHoldPixels[x] = max;//setting the pixels that arent black into white

}

else

{

threshHoldPixels[x] = 0;

}

}

}

blurredPixels.convertToType(FREE\_IMAGE\_TYPE::FIT\_BITMAP);

blurredPixels.convertTo24Bits();

blurredPixels.save("../Images/stage2\_threshold.png");

cout << "Binary Threshold has been applied and saved." << endl;

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << duration.count() << " Milliseconds for part2 sequential." << endl;

}

void part2TBB()

{

auto start = high\_resolution\_clock::now();

const int kHeight = 5;

const int kWidth = 5;

float sigma = 10.25f;

float kernelGaussian[kWidth][kHeight];

double sum = 0.0;

//creating guassian kernal

for (int i = 0; i < kWidth; i++)

{

for (int j = 0; j < kHeight; j++)

{

kernelGaussian[i][j] = exp(-(i \* i + j \* j) / (2 \* sigma \* sigma)) / (2 \* 3.142f \* sigma \* sigma);

sum += kernelGaussian[i][j];

}

}

for (int i = 0; i < kWidth; i++)

{

for (int j = 0; j < kHeight; j++)

{

kernelGaussian[i][j] /= sum;

}

}

fipImage inputImageStage1;

inputImageStage1.load("../Images/stage1\_combined.png");

auto height = inputImageStage1.getHeight();

auto width = inputImageStage1.getWidth();

inputImageStage1.convertToFloat();

float\* inputBuffer = (float\*)inputImageStage1.accessPixels();

fipImage outputImageStage2Blurred;

outputImageStage2Blurred = fipImage(FIT\_FLOAT, width, height, 32);

float\* outputBuffer = (float\*)outputImageStage2Blurred.accessPixels();

const int border = (kWidth - 1) / 2;

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>((uint64\_t)border, kHeight - border, (uint64\_t)border, kWidth - border), [=](const blocked\_range2d<uint64\_t, uint64\_t>& r)//parallel for function to model the range of the image

{

auto yBegin = r.rows().begin();//getting the start and ends for each axes

auto yEnd = r.rows().end();

auto xBegin = r.cols().begin();

auto xEnd = r.cols().end();

for (int y = yBegin; y < height - yEnd; y++)

{

for (int x = xBegin; x < width - xEnd; x++)

{

for (int i = -border; i < border; i++)

{

for (int j = -border; j < border; j++)

{

outputBuffer[y \* width + x] = outputBuffer[y \* width + x] + inputBuffer[(y - i) \* width + (x - j)] \* kernelGaussian[i + border][j + border];

}

}

}

}

});

outputImageStage2Blurred.convertToType(FREE\_IMAGE\_TYPE::FIT\_BITMAP);

outputImageStage2Blurred.convertTo24Bits();

outputImageStage2Blurred.save("../Images/stage2\_blurredTBB.png");

cout << "TBB: Blurring has been completed and saved." << endl;

fipImage thresholdPixels;

thresholdPixels.load("../Images/stage2\_blurredTBB.png");

auto tWidth = thresholdPixels.getWidth();

auto tHeight = thresholdPixels.getHeight();

thresholdPixels.convertToFloat();

float\* outputBuffer1 = (float\*)thresholdPixels.accessPixels();

fipImage blurredPixels;

blurredPixels = fipImage(FIT\_FLOAT, width, height, outputImageStage2Blurred.getBitsPerPixel());

blurredPixels.convertToFloat();

float\* threshHoldPixels = (float\*)blurredPixels.accessPixels();

float threshold = 0.0f;

float max = 255.0f;

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>((uint64\_t)0, tHeight - 0, (uint64\_t)0, tWidth - 0), [&](const blocked\_range2d<uint64\_t, uint64\_t>& r)//parallel for function which uses the variables instead of copying

{

auto yBegin = r.rows().begin();

auto yEnd = r.rows().end();

auto xBegin = r.cols().begin();

auto xEnd = r.cols().end();

for (int y = yBegin; y < yEnd; y++)

{

for (int x = xBegin; x < xEnd - 1; x++, outputBuffer1++, threshHoldPixels++)

{

float currentPixel = outputBuffer1[x];

if (currentPixel > threshold)

{

threshHoldPixels[x] = max;

}

else

{

threshHoldPixels[x] = 0;

}

}

}

});

blurredPixels.convertToType(FREE\_IMAGE\_TYPE::FIT\_BITMAP);

blurredPixels.convertTo24Bits();

blurredPixels.save("../Images/stage2\_thresholdTBB.png");

cout << "TBB: Binary Threshold has been applied and saved." << endl;

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << duration.count() << " Milliseconds for part2 parallel (TBB)." << endl;

}

void part3()

{

auto start = high\_resolution\_clock::now();

fipImage inputRenderTop1;

inputRenderTop1.load("../Images/render\_top\_1.png");

RGBQUAD\* colours = new RGBQUAD();

RGBQUAD\* colours1 = new RGBQUAD();

RGBQUAD\* colours2 = new RGBQUAD();

fipImage inputThreshold;

inputThreshold.load("../Images/stage2\_threshold.png");

int height = inputThreshold.getHeight();

int width = inputThreshold.getWidth();

float total = height \* width;

float total\_white = 0;

for (int y = 0; y < height; y++)

{

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

{

inputThreshold.getPixelColor(x, y, colours);

if (colours->rgbRed == 255 && colours->rgbGreen == 255 && colours->rgbBlue == 255)//for loop to go through the pixels with if statement making sure the pixel is white then adding a value to my float

{

total\_white++;

}

}

}

float percent = total\_white / total \* 100;//calculating the oercent of white pixels

cout << "The percentage of white pixels is = " << percent << "%" << endl;

for (int y = 0; y < height; y++)

{

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

{

inputThreshold.getPixelColor(x, y, colours);//getting pixel xolours for both images

inputRenderTop1.getPixelColor(x, y, colours1);

if (colours->rgbRed == 255 && colours->rgbGreen == 255 && colours->rgbBlue == 255)

{

colours2->rgbRed = 255 - colours1->rgbRed;//setting the colours for the pixels in my output image

colours2->rgbGreen = 255 - colours1->rgbGreen;

colours2->rgbBlue = 255 - colours1->rgbBlue;

inputRenderTop1.setPixelColor(x, y, colours2);

}

}

}

if (inputRenderTop1.save("../Images/stage3\_inverted.png"))//if statement to save because i was having trouble with saving as it wouldn't save

{

cout << "Inverted image is saved as stage3\_inverted.png" << endl;

}

else

{

cout << "Image was not saved!" << endl;

}

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << duration.count() << " Milliseconds for part3." << endl;

}

void part3TBB()

{

auto start = high\_resolution\_clock::now();

fipImage inputRenderTop1;

inputRenderTop1.load("../Images/render\_top\_1.png");

fipImage inputThreshold;

inputThreshold.load("../Images/stage2\_threshold.png");

int height = inputThreshold.getHeight();

int width = inputThreshold.getWidth();

float total = height \* width;

float total\_white = 0;

mutex mutex;

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>((uint64\_t)0, height - 0, (uint64\_t)0, width - 0), [&](const blocked\_range2d<uint64\_t, uint64\_t>& r)//parallel for function which uses the variables instead of copying

{

auto yBegin = r.rows().begin();

auto yEnd = r.rows().end();

auto xBegin = r.cols().begin();

auto xEnd = r.cols().end();

RGBQUAD\* colours = new RGBQUAD();

mutex.lock();

for (int y = yBegin; y < yEnd; y++)

{

for (int x = xBegin; x < xEnd; x++)

{

inputThreshold.getPixelColor(x, y, colours);

if (colours->rgbRed == 255 && colours->rgbGreen == 255 && colours->rgbBlue == 255)

{

total\_white++;

}

}

}

mutex.unlock();

});

float percent = total\_white / total \* 100;

cout << "TBB: The percentage of white pixels is = " << percent << "%" << endl;

parallel\_for(blocked\_range2d<uint64\_t, uint64\_t>((uint64\_t)0, height - 0, (uint64\_t)0, width - 0), [&](const blocked\_range2d<uint64\_t, uint64\_t>& r)//parallel for function which uses the variables instead of copying

{

auto yBegin = r.rows().begin();

auto yEnd = r.rows().end();

auto xBegin = r.cols().begin();

auto xEnd = r.cols().end();

RGBQUAD\* colours = new RGBQUAD();

RGBQUAD\* colours1 = new RGBQUAD();

RGBQUAD\* colours2 = new RGBQUAD();

for (int y = yBegin; y < yEnd; y++)

{

for (int x = xBegin; x < xEnd; x++)

{

inputThreshold.getPixelColor(x, y, colours);

inputRenderTop1.getPixelColor(x, y, colours1);

if (colours->rgbRed == 255 && colours->rgbGreen == 255 && colours->rgbBlue == 255)

{

colours2->rgbRed = 255 - colours1->rgbRed;

colours2->rgbGreen = 255 - colours1->rgbGreen;

colours2->rgbBlue = 255 - colours1->rgbBlue;

inputRenderTop1.setPixelColor(x, y, colours2);

}

}

}

});

if (inputRenderTop1.save("../Images/stage3\_invertedTBB.png"))

{

cout << "TBB: Inverted image is saved as stage3\_inverted.png" << endl;

}

else

{

cout << "TBB: Image was not saved!" << endl;

}

auto stop = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(stop - start);

cout << duration.count() << " Milliseconds for part3 parallel (TBB)." << endl;

}

int main()

{

int nt = task\_scheduler\_init::default\_num\_threads();

task\_scheduler\_init T(nt);

//Part 1 (Image Comparison): -----------------DO NOT REMOVE THIS COMMENT----------------------------//

part1();

part1multi();

//Part 2 (Blur & post-processing): -----------DO NOT REMOVE THIS COMMENT----------------------------//

part2();

part2TBB();

//Part 3 (Image Mask): -----------------------DO NOT REMOVE THIS COMMENT----------------------------//

part3();

part3TBB();

return 0;

}