Image Processing using TBB

**Author**: Matthew Fowler

**Student ID**: 17025958

**Module**: Parallel and Concurrent Programming

**Module Code**: CS3S666

**Module lecturers**: Marius Miknis, Simon Payne

# Speed Comparison

# Appendix A – Images & Screenshots

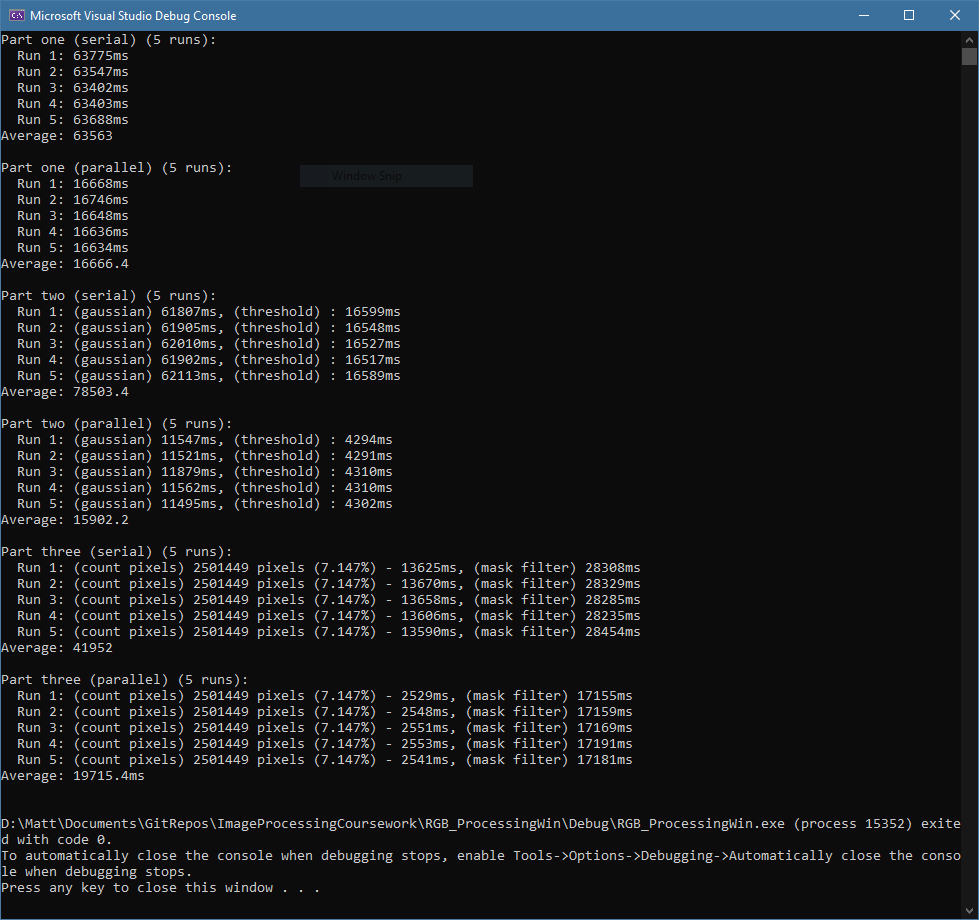


Figure : screenshot of speed tests

# Appendix B – Code (updated 27/2/20 16:41)

#include <iostream>

#include <vector>

//Thread building blocks library

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

//Free Image library

#include <FreeImagePlus.h>

//My includes

#include <functional>

#include <thread>

#include <chrono>

#include <cmath>

#include <tbb/parallel\_for.h>

#include <tbb/blocked\_range.h>

#include <tbb/blocked\_range2d.h>

#include <tbb/parallel\_reduce.h>

//Defines

#define M\_PIf 3.14159265358979f // reduces computation overhead when using pi.

using namespace std;

using namespace tbb;

// -------------------- TYPES --------------------

struct pixel\_rgb

{

alignas(1) unsigned char r = 0;

alignas(1) unsigned char g = 0;

alignas(1) unsigned char b = 0;

};

// generic functor parent

class StencilTask

{

public:

virtual float operator()(float x, float y) = 0;

};

// Gaussian functor

// use constructor to set Gaussian parameters 'sigma' and 'kernal pre-load radius' e.g. GaussianBlur MyGaussian = GaussianBlur(0.8f, 2)

// once instantiated, object can be called as if it was a function e.g. MyGaussian(x\_coord, y\_coord)

// NOTE: 'kradius' is the number of pixels away from the origin that the mask is applied to, e.g. kradius=2 means a 5x5 matrix

class GaussianBlur : public StencilTask

{

private:

float sigma;

int kradius;

vector<vector<float>> kernel;

// Pre-load mask matrix to avoid computing the same value multiple times.

// With kradius=0 no values will be pre-loaded and everything is computed dynamically.

void loadKernel(int kradius)

{

// x and y start from -kradius and finish at kradius inclusively.

for (int x = -(kradius); x <= kradius; x++)

{

vector<float> curRow;

for (int y = -(kradius); y <= kradius; y++)

{

curRow.push\_back(1.0f / (2.0f \* M\_PIf \* pow(sigma, 2)) \* exp(-((pow(x, 2) + pow(y, 2)) / (2.0f \* pow(sigma, 2.0f)))));

}

kernel.push\_back(curRow);

}

}

public:

// 'kradius' is the number of values to the left and right of the origin in the pre-loaded mask matrix.

// It is not a hard limit on the number of computable values.

GaussianBlur(float sigma, int kradius = 0)

{

this->sigma = sigma;

this->kradius = kradius;

this->loadKernel(kradius);

}

float operator()(float x, float y)

{

if (abs(x) <= kradius && abs(y) <= kradius) return kernel[x + kradius][y + kradius]; //value HAS been pre-loaded

else return 1.0f / (2.0f \* M\_PIf \* pow(sigma, 2)) \* exp(-((pow(x, 2) + pow(y, 2)) / (2.0f \* pow(sigma, 2)))); //value has NOT been pre-loaded

}

};

// -------------------- PART 1 --------------------

// Combines the pixels of one image with the pixels of another. Both images must have the same dimensions.

bool CombineImagesSerial(char\* inImagePathA, char\* inImagePathB, char\* outImagePath, function<pixel\_rgb(pixel\_rgb, pixel\_rgb)> fn)

{

// Load input images from disk

fipImage imgInA;

fipImage imgInB;

imgInA.load(inImagePathA);

imgInB.load(inImagePathB);

imgInA.convertTo24Bits();

imgInB.convertTo24Bits();

// Both input images must have the same dimensions

if (imgInA.getWidth() != imgInB.getWidth() || imgInA.getHeight() != imgInB.getHeight()) return false;

// Image dimensions

unsigned int width = imgInA.getWidth();

unsigned int height = imgInA.getHeight();

unsigned int numPixels = width \* height;

// Iterate over each pixel and apply the given function

pixel\_rgb\* aPointer = (pixel\_rgb\*)imgInA.accessPixels(); // also acts as the output image

pixel\_rgb\* bPointer = (pixel\_rgb\*)imgInB.accessPixels();

for (unsigned int pixel = 0; pixel < numPixels; pixel++, aPointer++, bPointer++)

{

\*aPointer = fn(\*aPointer, \*bPointer);

}

// Save output image to disk

return imgInA.save(outImagePath);

}

// A component of ComineImagesParallel. Combines only a sub-section of the given images.

void CombineSubImage(pixel\_rgb\* aPointer, pixel\_rgb\* bPointer, pixel\_rgb\* outPointer, unsigned int numIterations, function<pixel\_rgb(pixel\_rgb, pixel\_rgb)> fn)

{

for (unsigned int pixel = 0; pixel < numIterations; pixel++, aPointer++, bPointer++, outPointer++)

{

\*outPointer = fn(\*aPointer, \*bPointer);

}

}

// Combines the pixels of one image with the pixels of another. Both images must have the same dimensions.

bool CombineImagesParallel(char\* inImagePathA, char\* inImagePathB, char\* outImagePath, uint64\_t numThreads, function<pixel\_rgb(pixel\_rgb, pixel\_rgb)> fn)

{

// Load input images from disk as FreeImagePlus images

fipImage imgInA;

fipImage imgInB;

imgInA.load(inImagePathA);

imgInB.load(inImagePathB);

imgInA.convertTo24Bits();

imgInB.convertTo24Bits();

// Both input images must have the same dimensions

if (imgInA.getWidth() != imgInB.getWidth() || imgInA.getHeight() != imgInB.getHeight()) return false;

// Create an empty output image with the same dimensions as the inputs

unsigned int width = imgInA.getWidth();

unsigned int height = imgInA.getHeight();

unsigned int numPixels = width \* height;

fipImage imgOut(FIT\_BITMAP, width, height, 24);

// Create threads to process smaller sub-images

vector<thread> threads;

unsigned int stepsize = numPixels / numThreads;

unsigned int remainder = numPixels % numThreads;

pixel\_rgb\* aPointer = (pixel\_rgb\*)imgInA.accessPixels();

pixel\_rgb\* bPointer = (pixel\_rgb\*)imgInB.accessPixels();

pixel\_rgb\* outPointer = (pixel\_rgb\*)imgOut.accessPixels();

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

{

if (i == 0)

{

threads.push\_back(thread(CombineSubImage, aPointer, bPointer, outPointer, stepsize + remainder, fn));

aPointer += stepsize + remainder;

bPointer += stepsize + remainder;

outPointer += stepsize + remainder;

}

else

{

threads.push\_back(thread(CombineSubImage, aPointer, bPointer, outPointer, stepsize, fn));

aPointer += stepsize;

bPointer += stepsize;

outPointer += stepsize;

}

}

// Wait for the threads to finish executing

for (auto& thread : threads)

{

thread.join();

}

// Save output image to disk

return imgOut.save(outImagePath);

}

// -------------------- PART 2 --------------------

// Blurs the input image using the stencil pattern. The coefficients are defined by the given functor.

bool BlurImageSerial(char\* inImagePath, char\* outImagePath, int kradius, StencilTask& STask)

{

// Load input image from disk into memory

fipImage imgIn;

imgIn.load(inImagePath);

imgIn.convertToFloat();

// Load empty output image into memory

unsigned int width = imgIn.getWidth();

unsigned int height = imgIn.getHeight();

unsigned int numPixels = width \* height;

fipImage imgOut(FIT\_FLOAT, width, height, 32);

// Setup variables for accessing memory

float\* inPointer = (float\*)imgIn.accessPixels();

float\* outPointer = (float\*)imgOut.accessPixels();

float sum;

unsigned int stencilIndex = 0;

// Iterate over all pixels linearly

for (unsigned int yOrigin = 0; yOrigin < height; yOrigin++)

{

for (unsigned int xOrigin = 0; xOrigin < width; xOrigin++, outPointer++)

{

sum = 0.0f;

for (int yStencil = -(kradius); yStencil <= kradius; yStencil++)

{

// Make sure that the stencil's y-coord is within range. If not, snap to the nearest border pixel.

int absoluteYStencil = (int)yOrigin + yStencil;

if (absoluteYStencil < 0) absoluteYStencil = 0;

else if (absoluteYStencil >= height) absoluteYStencil = height - 1;

for (int xStencil = -(kradius); xStencil <= kradius; xStencil++)

{

// Make sure that the stencil's x-coord is within range. If not, snap to the nearest border pixel.

int absoluteXStencil = (int)xOrigin + xStencil;

if (absoluteXStencil < 0) absoluteXStencil = 0;

else if (absoluteXStencil >= width) absoluteXStencil = width - 1;

stencilIndex = (absoluteYStencil \* width) + absoluteXStencil;

sum += inPointer[stencilIndex] \* STask(xStencil, yStencil);

}

}

\*outPointer = sum;

}

}

// Save output image to disk

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

imgOut.convertTo24Bits();

return imgOut.save(outImagePath);

}

// For each pixel in the input image, apply the given function to it's value.

bool ApplyToImageSerial(char\* inImagePath, char\* outImagePath, function<pixel\_rgb(pixel\_rgb)> fn)

{

// Load input images from disk as FreeImagePlus images

fipImage imgIn;

imgIn.load(inImagePath);

imgIn.convertTo24Bits();

// Create an empty output image with the same dimensions as the inputs

unsigned int width = imgIn.getWidth();

unsigned int height = imgIn.getHeight();

unsigned int numPixels = width \* height;

// Iterate over each pixel and apply the given function

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgIn.accessPixels();

for (unsigned int pixel = 0; pixel < numPixels; pixel++, inPointer++)

{

\*inPointer = fn(\*inPointer);

}

// Save output image to disk

return imgIn.save(outImagePath);

}

// Blurs the input image using the stencil pattern. The coefficients are defined by the given functor.

bool BlurImageParallel(char\* inImagePath, char\* outImagePath, int kradius, StencilTask& STask)

{

fipImage imgIn;

imgIn.load(inImagePath);

imgIn.convertToFloat();

// Load empty output image into memory

unsigned int width = imgIn.getWidth();

unsigned int height = imgIn.getHeight();

unsigned int numPixels = width \* height;

fipImage imgOut(FIT\_FLOAT, width, height, 32);

// Iterate over all pixels using TBB parallel\_for

float\* inPointer = (float\*)imgIn.accessPixels();

float\* outPointer = (float\*)imgOut.accessPixels();

// For each pixel in the input...

parallel\_for(blocked\_range2d<int>(0, (int)height, 128, 0, (int)width, 128), [&](const blocked\_range2d<int>& dim)

{

int xstart = dim.cols().begin();

int ystart = dim.rows().begin();

int xend = dim.cols().end();

int yend = dim.rows().end();

int xsum = 0;

int ysum = 0;

for (int x = xstart; x != xend; x++)

{

for (int y = ystart; y != yend; y++)

{

for (int kx = -kradius; kx <= kradius; kx++)

{

xsum = x + kx;

if (xsum < 0) xsum = 0;

if (xsum >= width) xsum = width - 1;

for (int ky = -kradius; ky <= kradius; ky++)

{

ysum = y + ky;

if (ysum < 0) ysum = 0;

if (ysum >= height) ysum = height - 1;

outPointer[(y \* width) + x] += inPointer[(ysum \* width) + xsum] \* STask(kx, ky);

}

}

}

}

});

// Save output image to disk

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

imgOut.convertTo24Bits();

return imgOut.save(outImagePath);

}

// For each pixel in the input image, apply the given function to it's value.

bool ApplyToImageParallel(char\* inImagePath, char\* outImagePath, function<pixel\_rgb(pixel\_rgb)> fn)

{

// Load input images from disk as FreeImagePlus images

fipImage imgIn;

imgIn.load(inImagePath);

imgIn.convertTo24Bits();

unsigned int numPixels = imgIn.getWidth() \* imgIn.getHeight();

// Iterate over each pixel and apply the given function

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgIn.accessPixels();

parallel\_for(blocked\_range<int>(0, (int)numPixels, 1024), [&](const blocked\_range<int>& range) {

for (int i = range.begin(); i < range.end(); i++)

{

inPointer[i] = fn(inPointer[i]);

}

});

// Save output image to disk

return imgIn.save(outImagePath);

}

// -------------------- PART 3 --------------------

// Returns the number of pixels that meet the criteria given.

int PixelsThatMeetCriteriaSerial(char\* inImagePath, function<bool(pixel\_rgb)> criteria)

{

// Load input images from disk as FreeImagePlus images

fipImage imgIn;

imgIn.load(inImagePath);

imgIn.convertTo24Bits();

unsigned int numPixels = imgIn.getWidth() \* imgIn.getHeight();

// Iterate over each pixel and count how many pixels meet the given criteria

int sum = 0;

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgIn.accessPixels();

for (unsigned int pixel = 0; pixel < numPixels; pixel++, inPointer++)

{

if (criteria(\*inPointer)) sum++;

}

return sum;

}

// Inverts the pixels in the input image, where the corresponding pixel in the mask meets the given condition.

bool MaskInvertSerial(char\* inImagePath, char\* maskImagePath, char\* outImagePath, function<bool(pixel\_rgb)> maskCondition)

{

// Load the input image and the mask from the disk

fipImage imgIn;

fipImage imgMask;

imgIn.load(inImagePath);

imgMask.load(maskImagePath);

imgIn.convertTo24Bits();

imgMask.convertTo24Bits();

// Both input images must have the same dimensions

if (imgIn.getWidth() != imgMask.getWidth() || imgIn.getHeight() != imgMask.getHeight()) return false;

unsigned int numPixels = imgIn.getWidth() \* imgIn.getHeight();

// For each pixel in the input, check that the corresponding pixel in the mask meets the condition

// If it does invert the pixel in the input image

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgIn.accessPixels();

pixel\_rgb\* maskPointer = (pixel\_rgb\*)imgMask.accessPixels();

for (unsigned int pixel = 0; pixel < numPixels; pixel++, inPointer++, maskPointer++)

{

if (maskCondition(\*maskPointer))

{

inPointer->r = 255 - inPointer->r;

inPointer->g = 255 - inPointer->g;

inPointer->b = 255 - inPointer->b;

}

}

// Save result to the disk

return imgIn.save(outImagePath);

}

// Returns the number of pixels that meet the criteria given.

int PixelsThatMeetCriteriaParallel(char\* inImagePath, function<bool(pixel\_rgb)> criteria)

{

// Load input images from disk as FreeImagePlus images

fipImage imgIn;

imgIn.load(inImagePath);

imgIn.convertTo24Bits();

int numPixels = imgIn.getWidth() \* imgIn.getHeight();

// Iterate over each pixel and count how many pixels meet the given criteria

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgIn.accessPixels();

int sum = parallel\_reduce(

blocked\_range<int>(0, numPixels, 1024),

0,

[&](const blocked\_range<int>& range, int initValue) {

for (int i = range.begin(); i != range.end(); i++)

{

if (criteria(inPointer[i])) initValue++;

}

return initValue;

},

[&](int a, int b) {

return a + b;

}

);

return sum;

}

// Inverts the pixels in the input image, where the corresponding pixel in the mask meets the given condition.

bool MaskInvertParallel(char\* inImagePath, char\* maskImagePath, char\* outImagePath, function<bool(pixel\_rgb)> maskCondition)

{

// Load the input image and the mask from the disk

fipImage imgIn;

fipImage imgMask;

imgIn.load(inImagePath);

imgMask.load(maskImagePath);

imgIn.convertTo24Bits();

imgMask.convertTo24Bits();

// Both input images must have the same dimensions

if (imgIn.getWidth() != imgMask.getWidth() || imgIn.getHeight() != imgMask.getHeight()) return false;

unsigned int numPixels = imgIn.getWidth() \* imgIn.getHeight();

// For each pixel in the input, check that the corresponding pixel in the mask meets the condition

// If it does invert the pixel in the input image

pixel\_rgb\* inPointer = (pixel\_rgb\*)imgIn.accessPixels();

pixel\_rgb\* maskPointer = (pixel\_rgb\*)imgMask.accessPixels();

parallel\_for(blocked\_range<int>(0, numPixels, 1024), [&](blocked\_range<int>& range) {

for (int p = range.begin(); p < range.end(); p++)

{

if (maskCondition(maskPointer[p]))

{

inPointer[p].r = 255 - inPointer[p].r;

inPointer[p].g = 255 - inPointer[p].g;

inPointer[p].b = 255 - inPointer[p].b;

}

}

});

// Save result to the disk

return imgIn.save(outImagePath);

}

// MAIN FUNCTION

int main()

{

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

task\_scheduler\_init T(nt);

// Image file paths

char IN\_TOP\_1[] = "../Images/render\_top\_1.png";

char IN\_TOP\_2[] = "../Images/render\_top\_2.png";

char IN\_BOTTOM\_1[] = "../Images/render\_bottom\_1.png";

char IN\_BOTTOM\_2[] = "../Images/render\_bottom\_2.png";

char OUT\_STAGE\_1\_TOP[] = "../Images/stage1\_top.png";

char OUT\_STAGE\_1\_BOTTOM[] = "../Images/stage1\_bottom.png";

char OUT\_STAGE\_1\_COMBINED[] = "../Images/stage1\_combined.png";

char OUT\_STAGE\_2\_BLURRED[] = "../Images/stage2\_blurred.png";

char OUT\_STAGE\_2\_THRESHOLD[] = "../Images/stage2\_threshold.png";

char OUT\_STAGE\_3[] = "../Images/stage3\_final.png";

// Number of times each stage will be executed

const int STAGE\_1\_SEQ\_ITERATIONS = 5;

const int STAGE\_1\_PAR\_ITERATIONS = 5;

const int STAGE\_2\_SEQ\_ITERATIONS = 5;

const int STAGE\_2\_PAR\_ITERATIONS = 5;

const int STAGE\_3\_SEQ\_ITERATIONS = 5;

const int STAGE\_3\_PAR\_ITERATIONS = 5;

std::chrono::steady\_clock::time\_point start;

std::chrono::steady\_clock::time\_point end;

float average = 0.0f;

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

// IF pixel 'a' is the same as pixel 'b', return a black pixel. Otherwise return a white one

auto and = [](pixel\_rgb a, pixel\_rgb b)->pixel\_rgb

{

pixel\_rgb output;

if (a.r == b.r && a.g == b.g && a.b == b.b)

{

output.r = 0;

output.g = 0;

output.b = 0;

}

else

{

output.r = 255;

output.g = 255;

output.b = 255;

}

return output;

};

// return a new pixel by adding half of the RGB values of pixel 'a' to half of pixel 'b'

auto sum = [](pixel\_rgb a, pixel\_rgb b)->pixel\_rgb

{

pixel\_rgb output;

output.r = (a.r / (unsigned char)2) + (b.r / (unsigned char)2);

output.g = (a.g / (unsigned char)2) + (b.g / (unsigned char)2);

output.b = (a.b / (unsigned char)2) + (b.b / (unsigned char)2);

return output;

};

// Part 1 sequential solution:

cout << "Part one (serial) (" << STAGE\_1\_SEQ\_ITERATIONS << " runs):" << endl;

for (int i = 0; i < STAGE\_1\_SEQ\_ITERATIONS; i++)

{

start = std::chrono::steady\_clock::now();

CombineImagesSerial(IN\_TOP\_1, IN\_TOP\_2, OUT\_STAGE\_1\_TOP, and);

CombineImagesSerial(IN\_BOTTOM\_1, IN\_BOTTOM\_2, OUT\_STAGE\_1\_BOTTOM, and);

CombineImagesSerial(OUT\_STAGE\_1\_TOP, OUT\_STAGE\_1\_BOTTOM, OUT\_STAGE\_1\_COMBINED, sum);

end = std::chrono::steady\_clock::now();

auto duration\_p1\_s = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << " Run " << i+1 << ": " << duration\_p1\_s << "ms" << endl;

average += (float)duration\_p1\_s;

}

if (STAGE\_1\_SEQ\_ITERATIONS > 0) average = average / (float)STAGE\_1\_SEQ\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

// Part 1 parallel solution:

cout << "Part one (parallel) (" << STAGE\_1\_PAR\_ITERATIONS << " runs):" << endl;

for (int i = 0; i < STAGE\_1\_PAR\_ITERATIONS; i++)

{

start = std::chrono::steady\_clock::now();

CombineImagesParallel(IN\_TOP\_1, IN\_TOP\_2, OUT\_STAGE\_1\_TOP, nt, and);

CombineImagesParallel(IN\_BOTTOM\_1, IN\_BOTTOM\_2, OUT\_STAGE\_1\_BOTTOM, nt, and);

CombineImagesParallel(OUT\_STAGE\_1\_TOP, OUT\_STAGE\_1\_BOTTOM, OUT\_STAGE\_1\_COMBINED, nt, sum);

end = std::chrono::steady\_clock::now();

auto duration\_p1\_p = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << " Run " << i+1 << ": " << duration\_p1\_p << "ms" << endl;

average += (float)duration\_p1\_p;

}

if (STAGE\_1\_PAR\_ITERATIONS > 0) average = average / (float)STAGE\_1\_PAR\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

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

// blur parameters

float sigma = 0.8f;

int kernal\_radius = 1;

// IF the given pixel is not black, return a white pixel. Otherwise return a black pixel

auto binaryThreshold = [](pixel\_rgb x)->pixel\_rgb {

pixel\_rgb white;

white.r = 255; white.g = 255; white.b = 255;

if (x.r != 0 || x.g != 0 || x.b != 0) return white;

else return x;

};

// Part 2 sequential solution:

cout << "Part two (serial) (" << STAGE\_2\_SEQ\_ITERATIONS << " runs): " << endl;

for (int i = 0; i < STAGE\_2\_SEQ\_ITERATIONS; i++)

{

start = std::chrono::steady\_clock::now();

BlurImageSerial(OUT\_STAGE\_1\_COMBINED, OUT\_STAGE\_2\_BLURRED, kernal\_radius, GaussianBlur(sigma, kernal\_radius));

end = std::chrono::steady\_clock::now();

auto duration\_p2\_s1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << " Run " << i+1 << ": (gaussian) " << duration\_p2\_s1 << "ms, ";

start = std::chrono::steady\_clock::now();

ApplyToImageSerial(OUT\_STAGE\_2\_BLURRED, OUT\_STAGE\_2\_THRESHOLD, binaryThreshold);

end = std::chrono::steady\_clock::now();

auto duration\_p2\_s2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(threshold) " << ": " << duration\_p2\_s2 << "ms" << endl;

average += (float)duration\_p2\_s1 + (float)duration\_p2\_s2;

}

if (STAGE\_2\_SEQ\_ITERATIONS > 0) average = average / (float)STAGE\_2\_SEQ\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

// Part 2 parallel solution:

cout << "Part two (parallel) (" << STAGE\_2\_PAR\_ITERATIONS << " runs): " << endl;

for (int i = 0; i < STAGE\_2\_PAR\_ITERATIONS; i++)

{

start = std::chrono::steady\_clock::now();

BlurImageParallel(OUT\_STAGE\_1\_COMBINED, OUT\_STAGE\_2\_BLURRED, kernal\_radius, GaussianBlur(sigma, kernal\_radius));

end = std::chrono::steady\_clock::now();

auto duration\_p2\_p1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << " Run " << i + 1 << ": (gaussian) " << duration\_p2\_p1 << "ms, ";

start = std::chrono::steady\_clock::now();

ApplyToImageParallel(OUT\_STAGE\_2\_BLURRED, OUT\_STAGE\_2\_THRESHOLD, binaryThreshold);

end = std::chrono::steady\_clock::now();

auto duration\_p2\_p2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(threshold) " << ": " << duration\_p2\_p2 << "ms" << endl;

average += (float)duration\_p2\_p1 + (float)duration\_p2\_p2;

}

if (STAGE\_2\_PAR\_ITERATIONS > 0) average = average / (float)STAGE\_2\_PAR\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

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

// IF pixel 'x' is white, return true. Otherwise return false

auto checkPixelIsWhite = [](pixel\_rgb x)->bool {

if (x.r == 255 && x.g == 255 && x.b == 255) return true;

else return false;

};

// Part 3 sequential solution:

cout << "Part three (serial) (" << STAGE\_3\_SEQ\_ITERATIONS << " runs): " << endl;

for (int i = 0; i < STAGE\_3\_SEQ\_ITERATIONS; i++)

{

cout << " Run " << i+1 << ": (count pixels) ";

start = std::chrono::steady\_clock::now();

int vals = PixelsThatMeetCriteriaSerial(OUT\_STAGE\_2\_THRESHOLD, checkPixelIsWhite);

end = std::chrono::steady\_clock::now();

cout << vals << " pixels (" << ((float)vals / (5000.0f \* 7000.0f)) \* 100.0f << "%) - ";

auto duration\_p3\_s1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << duration\_p3\_s1 << "ms, ";

start = std::chrono::steady\_clock::now();

MaskInvertSerial(IN\_TOP\_1, OUT\_STAGE\_2\_THRESHOLD, OUT\_STAGE\_3, checkPixelIsWhite);

end = std::chrono::steady\_clock::now();

auto duration\_p3\_s2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(mask filter) " << duration\_p3\_s2 << "ms" << endl;

average += (float)duration\_p3\_s1 + (float)duration\_p3\_s2;

}

if (STAGE\_3\_SEQ\_ITERATIONS > 0) average = average / (float)STAGE\_3\_SEQ\_ITERATIONS;

cout << "Average: " << average << endl << endl;

average = 0.0f;

// Part 3 parallel solution:

cout << "Part three (parallel) (" << STAGE\_3\_PAR\_ITERATIONS << " runs): " << endl;

for (int i = 0; i < STAGE\_3\_PAR\_ITERATIONS; i++)

{

cout << " Run " << i + 1 << ": (count pixels) ";

start = std::chrono::steady\_clock::now();

int vals = PixelsThatMeetCriteriaParallel(OUT\_STAGE\_2\_THRESHOLD, checkPixelIsWhite);

end = std::chrono::steady\_clock::now();

cout << vals << " pixels (" << ((float)vals / (5000.0f \* 7000.0f)) \* 100.0f << "%) - ";

auto duration\_p3\_p1 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << duration\_p3\_p1 << "ms, ";

start = std::chrono::steady\_clock::now();

MaskInvertParallel(IN\_TOP\_1, OUT\_STAGE\_2\_THRESHOLD, OUT\_STAGE\_3, checkPixelIsWhite);

end = std::chrono::steady\_clock::now();

auto duration\_p3\_p2 = chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

cout << "(mask filter) " << duration\_p3\_p2 << "ms" << endl;

average += (float)duration\_p3\_p1 + (float)duration\_p3\_p2;

}

if (STAGE\_3\_PAR\_ITERATIONS > 0) average = average / (float)STAGE\_3\_PAR\_ITERATIONS;

cout << "Average: " << average << "ms" << endl << endl;

average = 0.0f;

return 0;

}