# Image Processing Comparison Investigation & OpenCL Implementation

## Testing

To test the speed differences for the gaussian blur kernel using sequential code compared to parallel code, 3 different size images are going to be used to provide different levels of workloads. The 3 sizes are going to be small (1024x576), medium (1920x1080) and large (3840x2160). The machine used for testing is a running a Linux operating system, an i5-6200u processor and 8GB RAM. The processor has a maximum number of 4 threads.

The first series of tests will be to test the speed differences between sequential and parallel code using the 3 images size mentioned above. The parallel code will be tested on 1,2 and 4 threads to find out how the number of threads available affects the speed differences on the code. Speeds for all tests are recorded in microseconds and each test is run 5 times to get the average running speed. The following tests are using a sigma value of 3.0 and a kernel size of 9.

The table below shows the speeds for the sequential and parallel code using the small sized image.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Sequential | 645641ms | 637742ms | 636280ms | 641491ms | 638182ms | 639867ms |
| Parallel – 1 Thread | 907239ms | 892144ms | 895948ms | 894446ms | 893643ms | 896684ms |
| Parallel – 2 Threads | 527915ms | 521719ms | 521396ms | 523725ms | 518338ms | 522619ms |
| Parallel – 4 Threads | 502461ms | 496246ms | 495889ms | 497363ms | 495790ms | 497550ms |

The table below shows the speeds for the sequential and parallel code using the medium sized image.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Sequential | 2312181ms | 2193566ms | 2187889ms | 2189004ms | 2187187ms | 2213965ms |
| Parallel – 1 Thread | 3350278ms | 3124937ms | 3109604ms | 3058028ms | 3110644ms | 3150698ms |
| Parallel – 2 Threads | 1788415ms | 1791680ms | 1782775ms | 1785071ms | 1784041ms | 1786396ms |
| Parallel – 4 Threads | 1730743ms | 1729462ms | 1722026ms | 1717285ms | 1729510ms | 1725805ms |

The table below shows the speeds for the sequential and parallel code using the large sized image.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Sequential | 8760897ms | 8760406ms | 8767431ms | 8769369ms | 8760251ms | 8763671ms |
| Parallel – 1 Thread | 12723125ms | 12285836ms | 12342418ms | 12257957ms | 12303469ms | 12382561ms |
| Parallel – 2 Threads | 7222599ms | 7177836ms | 7202943ms | 7177489ms | 7168424ms | 7189858ms |
| Parallel – 4 Threads | 6907417ms | 6892877ms | 6885931ms | 6888548ms | 6884061ms | 6891767ms |

To investigate the impact of step size has on parallel code, the large sized image is going to be used and the step sizes 5, 10, 50, 100 and 500 are going to be used. The test is going to be ran on 1,2 and 4 threads. Each test will run 5 times for an average speed to be collected. The following tests are using a sigma value of 3.0 and a kernel size of 9.

The table below shows the speeds for the parallel code with a step size of 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Parallel – 1 Thread | 12275894ms | 12292137ms | 12457011ms | 12498574ms | 12273591ms | 12359441ms |
| Parallel – 2 Threads | 7198727ms | 7168736ms | 7166769ms | 7254115ms | 7169376ms | 7191545ms |
| Parallel – 4 Threads | 6892239ms | 6873466ms | 6878821ms | 6880986ms | 6885518ms | 6882206ms |

The table below shows the speeds for the parallel code with a step size of 10.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Parallel – 1 Thread | 12410374ms | 12304581ms | 12369139ms | 12359306ms | 12280556ms | 12344791ms |
| Parallel – 2 Threads | 7174396ms | 7207777ms | 7153585ms | 7201764ms | 7218988ms | 7191302ms |
| Parallel – 4 Threads | 6889309ms | 6955241ms | 6888302ms | 6868019ms | 6893318ms | 6898838ms |

The table below shows the speeds for the parallel code with a step size of 50.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Parallel – 1 Thread | 12313985ms | 12327397ms | 12268342ms | 12319955ms | 12387955ms | 12323527ms |
| Parallel – 2 Threads | 7175945ms | 7180161ms | 7163207ms | 7213908ms | 7220379ms | 7190720ms |
| Parallel – 4 Threads | 6891522ms | 6898811ms | 6877509ms | 6878347ms | 6880293ms | 6885296ms |

The table below shows the speeds for the parallel code with a step size of 100.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Parallel – 1 Thread | 12318169ms | 12272835ms | 12267467ms | 12468889ms | 12305510ms | 12326574ms |
| Parallel – 2 Threads | 7218606ms | 7191595ms | 7164845ms | 7271459ms | 7165592ms | 7202419ms |
| Parallel – 4 Threads | 6902586ms | 6889488ms | 6888645ms | 6928252ms | 6898536ms | 6901501ms |

The table below shows the speeds for the parallel code with a step size of 500.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Parallel – 1 Thread | 12284568ms | 12362074ms | 12260914ms | 12573225ms | 12288924ms | 12353941ms |
| Parallel – 2 Threads | 7187171ms | 7201243ms | 7239870ms | 7237829ms | 7220444ms | 7217311ms |
| Parallel – 4 Threads | 6935785ms | 6987147ms | 7048335ms | 6915421ms | 7108660ms | 6999074ms |

## Results Discussion

Firstly, the results of the speed differences are going to be discussed. Overall, running a gaussian blur on an image sequentially was slower compared to running a gaussian blur using parallel code on 4 threads. In all 3 images size, running the blur kernel in parallel with just one thread available causes the code to run significantly slower.

The table below shows the difference in speed between the parallel code compared to the sequential code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Small Image | Medium Image | Large Image | Average Difference |
| 1 Thread | 40.14% Slower | 42.31% Slower | 41.29% Slower | 41.25% Slower |
| 2 Threads | 18.32% Faster | 19.31% Faster | 17.96% Faster | 18.53% Faster |
| 4 Threads | 22.24% Faster | 22.05% Faster | 21.36% Faster | 21.88% Faster |

Even through the image size were different, the speed differences using parallel code compared to sequential are within a few percent margin of each other meaning that regardless of the image size, similar speed differences were achieved. Running the parallel version of the gaussian blur on a single thread resulted in the code running on average 41.25% slower than the sequential blur. The 2 threaded parallel code and the 4 threaded parallel code ran 18.53% and 21.88% faster respectively with 4 threads obtain a small performance increase over 2 threads. These results show that running parallel code when multiple threads are available is beneficial and will provide a significant performance increase over sequential code. However, if parallel code is running on a system with only 1 thread available, the code will run far slower than the normal sequential code.

The next result to discuss is the speed differences between the different step sizes when running parallel code. Overall, these results show that the step size needs to be chosen carefully depending on the task the code needs to perform and the number of threads available to that task.

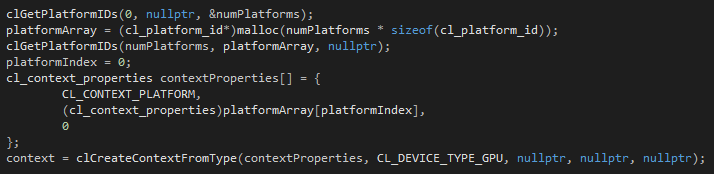
The table below shows the difference in speed between step sizes 5, 10, 50, 100 and 500 compared to the sequential code speed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Step Size 5 | Step Size 10 | Step Size 50 | Step Size 100 | Step Size 500 |
| 1 Thread | 41.03% Slower | 40.86% Slower | 40.62% Slower | 40.66% Slower | 40.97% Slower |
| 2 Threads | 17.94% Faster | 17.94% Faster | 17.95% Faster | 17.82% Faster | 17.65% Faster |
| 4 Threads | 21.47% Faster | 21.28% Faster | 21.43% Faster | 21.25% Faster | 20.14% Faster |

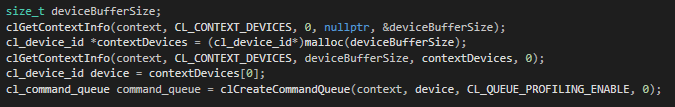
These results show a mixed reaction to the step size depending on the number of threads that are available to the task. If only 1 thread is available, the ideal step size for the program is around 50-100 to gain a small performance boost however it is worth noting that the performance increase in this case is significantly lower would barely be noticeable unless testing the performance. The absolute worst case for the code running on 1 thread would be an extremely low step size because it would be processing each chunk of the task in tiny pieces and calculating each step would create an overhead. For 2 or 4 threads however, A lower step size would be more optimal as each of the threads available can split and more the task more equally when the task is split up into more chunks. For multi-threaded code, a high step size would mean each of the threads are given a large chunk of the task at once which means that the task is likely to not be more split up between each of the threads.

## OpenCL Implementation

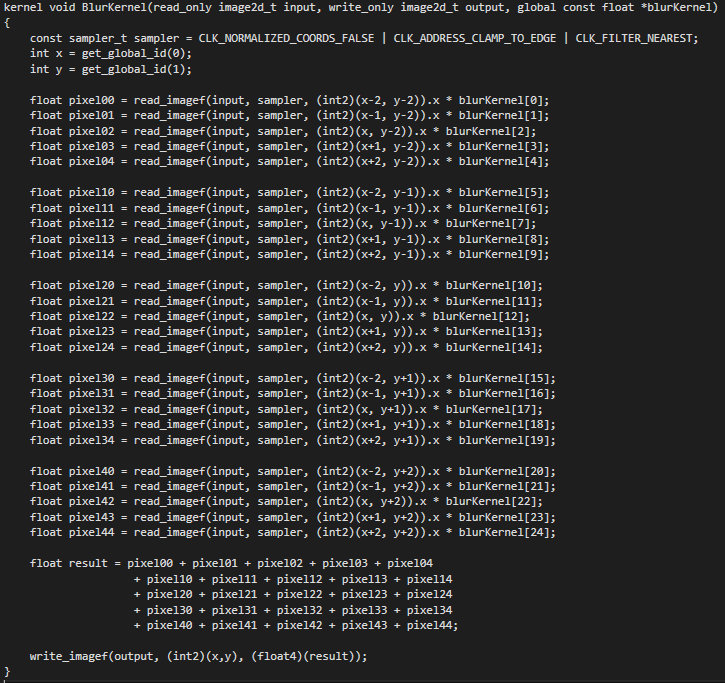
The TBB implantation of this code runs a couple task, Firstly the blur kernel is calculated and normalised based on the sigma and kernel size values, then the kernel is applied to each pixel in the image. For an OpenCL Implementation, a kernel of a set size is going to be used, but the sigma value of that kernel is changeable and calculated using normal CPP code. The reason for this change is that it makes it simpler to write and understand the OpenCL kernel.



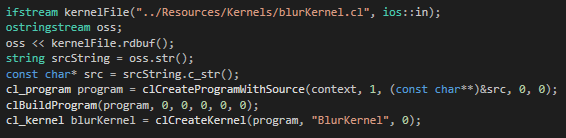
To create an Open CL version of the gaussian blur, An Open CL context must first be created. The context is created by checking which devices are available to use and makes a context based on the platform selected. A context is used to hold and manage all items that are associated with the current task such as memory objects, kernels and command queues.



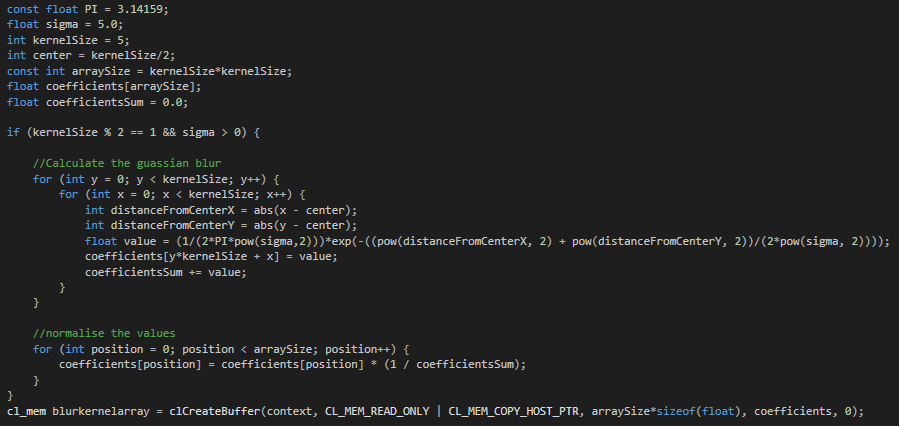
The next part of the program that needs to be created is the command queue. A command queue holds all the tasks at need to run on the platform. A command queue is created by retrieving the selected device from the context and creating a command queue for the given device.



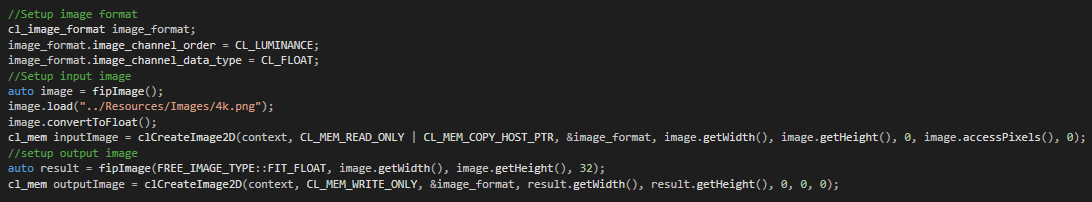
Now that a context and a command queue has been created, A kernel needs to be created next. The kernel shown above is the part of the program that will be ran on the compute units of the given platform to perform the gaussian blur on an image. This kernel will take an input image, apply the blur coefficients to the given pixel and return the result to the output image.



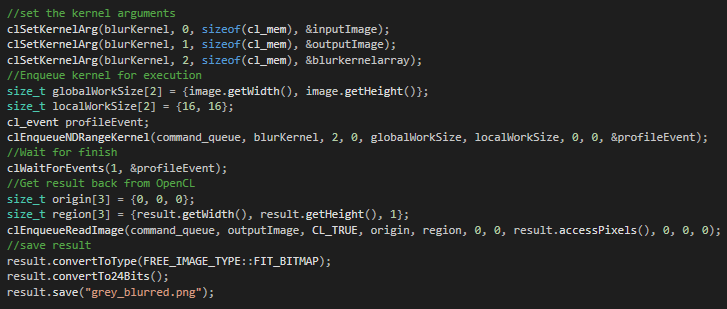
Now that the kernel is ready, A program object and a kernel object must be created. The program object is created by reading the kernel file into a string stream and builds a program based on the current context and the kernel it has retrieved from the string stream. A kernel object is created from the program object passing in the name of the kernel that we want to run which in this case is “BlurKernel”.



To calculate the blur matrix, the sigma value and kernel size are used in the gaussian blur equation to work out the values of each coefficient. The matrix is then normalized as the sum of the matrix should be equal to 1 because if the sum is higher than 1, the image would become darker. Once the matrix is calculated, an Open CL memory object is created and the coefficients matrix is passed into it, this is to make sure that the matrix calculated is available to the kernel.



The next memory objects to create are the input and output images. Before setting up the image objects, the image format must first be defined. Here we set the channel order to CL\_LUMINANCE and the datatype to CL\_FLOAT. This channel order makes the image grayscale and the data type simply specifies to the program that the values being read are going to be floats. The input image is then loaded and converted to floats also an output image is setup to hold the resulting image. 2 memory objects are created, the first one is for the input image and is a read only object which means the kernel can’t change the input image and the second object is for the output image which has write permissions.



Now that all the required objects have been created, the kernel can be run. Firstly, the kernel arguments are set to the 3 memory objects we created; these act as the parameters for the kernel function. The kernel is then enqueued to the command queue with a global work size equal to the size of the image and a local work size of 16x16. The program will then wait for the kernel to finish and get the result back from the kernel to save as an image file.

The following tests were called out with a large image, a kernel size of 5 and a sigma value of 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Sequential | 4130313ms | 4130549ms | 4131235ms | 4226046ms | 4133377ms | 4150304ms |
| Parallel – 1 Thread | 5306425ms | 5299260ms | 5475831ms | 5435886ms | 5507173ms | 5404915ms |
| Parallel – 2 Threads | 3730349ms | 3673740ms | 3733286ms | 3645876ms | 3694678ms | 3695586ms |
| Parallel – 4 Threads | 3649494ms | 3600111ms | 3613194ms | 3745786ms | 3598764ms | 3641470ms |
| Open CL W/ Integrated Graphics | 2446248ms | 2439711ms | 2436075ms | 2416242ms | 2420805ms | 2431816ms |
| Open CL W/ Nvidia 940M 2GB GPU | 2169970ms | 2167869ms | 2143319ms | 2143573ms | 2144006ms | 2153747ms |

As shown in the graph above, The Open CL version of the code running on integrated graphics is 41% faster than the sequential code and 33% faster than the parallel code running on the maximum number of threads. However, Running the Open CL code on an actual GPU (In this case, a Nvidia 940M with 2GB memory), The code runs 48% faster than the sequential code and 41% faster than the parallel code on 4 threads. These results are expected because graphics cards are better at running parallel tasks more efficiently than CPUs due to the higher number of compute cores.

## Source Code

Gaussian Blur

#include <iostream>

#include <vector>

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

#include <tbb/parallel\_for.h>

#include <tbb/blocked\_range.h>

#include <tbb/blocked\_range2d.h>

#include <FreeImagePlus.h>

#include <cmath>

#include <chrono>

using namespace std;

using namespace tbb;

void sequentialGaussianBlur(){

const float PI = 3.14159;

//Calculate the kernel

float sigma = 5.0;

int kernelSize = 5;

int center = kernelSize/2;

vector<float> kernel;

kernel.resize(kernelSize\*kernelSize);

float kernelSum = 0.0;

if (kernelSize % 2 == 1 && sigma > 0) {

for (int y = 0; y < kernelSize; y++) {

for (int x = 0; x < kernelSize; x++) {

int distanceFromCenterX = abs(x - center);

int distanceFromCenterY = abs(y - center);

float value = (1/(2\*PI\*pow(sigma,2)))\*exp(-((pow(distanceFromCenterX, 2) + pow(distanceFromCenterY, 2))/(2\*pow(sigma, 2))));

kernel[y\*kernelSize + x] = value;

kernelSum += value;

}

}

for (int position = 0; position < kernel.size(); position++) {

kernel[position] = kernel[position] \* (1 / kernelSum);

}

}

//load image from file and convert into a float dataset

fipImage inputImage;

inputImage.load("../Images/576p.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

//setup output image array

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

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

//apply the gaussian blur

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

{

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

{

for(int positionY = 0; positionY < kernelSize; positionY++)

{

for(int positionX = 0; positionX < kernelSize; positionX++)

{

int offsetX = positionX - center;

int offsetY = positionY - center;

if(((x + offsetX) > 0)&&((x + offsetX) < width)&&((y + offsetY) > 0)&&((y + offsetY) < height)){

outputBuffer[y\*width+x] += inputBuffer[(y+offsetY)\*width+(x+offsetX)] \* kernel[positionY\*kernelSize+positionX];

}

}

}

}

}

//save result

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

outputImage.convertTo24Bits();

outputImage.save("grey\_blurred.png");

}

void parallelGaussianBlur(void){

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

task\_scheduler\_init T(nt);

const float PI = 3.14159;

//Calculate the kernel

float sigma = 5.0;

int kernelSize = 5;

int center = kernelSize/2;

vector<float> kernel;

kernel.resize(kernelSize\*kernelSize);

float kernelSum = 0.0;

if (kernelSize % 2 == 1 && sigma > 0) {

parallel\_for(

blocked\_range2d<int, int>(0, kernelSize, 0, kernelSize),

[&](const blocked\_range2d<int, int>& range) {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

for (int y = y\_start; y < y\_end; y++) {

for (int x = x\_start; x < x\_end; x++) {

int distanceFromCenterX = abs(x - center);

int distanceFromCenterY = abs(y - center);

float value = (1/(2\*PI\*pow(sigma,2)))\*exp(-((pow(distanceFromCenterX, 2) + pow(distanceFromCenterY, 2))/(2\*pow(sigma, 2))));

kernel[y\*kernelSize + x] = value;

kernelSum += value;

}

}

}

);

parallel\_for(

blocked\_range<int>(0,kernel.size()),

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

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

kernel[position] = kernel[position] \* (1 / kernelSum);

}

}

);

}

//load image from file and convert into a float dataset

fipImage inputImage;

inputImage.load("../Images/576p.png");

inputImage.convertToFloat();

auto width = inputImage.getWidth();

auto height = inputImage.getHeight();

const float\* const inputBuffer = (float\*)inputImage.accessPixels();

//setup output image array

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

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

parallel\_for(

blocked\_range2d<int, int>(0, height, 0, width),

[&](const blocked\_range2d<int, int>& range) {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

//apply the gaussian blur

for (int y = y\_start; y < y\_end; y++) {

for (int x = x\_start; x < x\_end; x++) {

for (int positionY = 0; positionY < kernelSize; positionY++) {

for (int positionX = 0; positionX < kernelSize; positionX++) {

int offsetX = positionX - center;

int offsetY = positionY - center;

if(((x + offsetX) > 0)&&((x + offsetX) < width)&&((y + offsetY) > 0)&&((y + offsetY) < height)) {

outputBuffer[y \* width + x] += inputBuffer[(y + offsetY) \* width + (x + offsetX)] \*

kernel[positionY \* kernelSize + positionX];

}

}

}

}

}

}

);

//save result

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

outputImage.convertTo24Bits();

outputImage.save("grey\_blurred.png");

}

int main() {

auto t0 = std::chrono::high\_resolution\_clock::now();

sequentialGaussianBlur();

auto t1 = std::chrono::high\_resolution\_clock::now();

auto seqDuration = std::chrono::duration\_cast<std::chrono::microseconds>(t1 - t0);

cout << "Sequential Time: " << seqDuration.count() << " Microseconds" << endl;

auto t3 = std::chrono::high\_resolution\_clock::now();

parallelGaussianBlur();

auto t4 = std::chrono::high\_resolution\_clock::now();

auto parDuration = std::chrono::duration\_cast<std::chrono::microseconds>(t4 - t3);

cout << "Parallel Time: " << parDuration.count() << " Microseconds" << endl;

return 0;

}

RGB Processing

#include <iostream>

#include <FreeImagePlus.h>

#include <vector>

#include <random>

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

#include <tbb/parallel\_for.h>

#include <tbb/parallel\_reduce.h>

#include <tbb/blocked\_range.h>

#include <tbb/blocked\_range2d.h>

#include <chrono>

using namespace std;

using namespace tbb;

void parallelRGBProcessing(void) {

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

task\_scheduler\_init T(nt);

// setup input image 1 array and extract the color data

fipImage inputImage1;

inputImage1.load("../Images/render\_1.png");

int image1\_width = inputImage1.getWidth();

int image1\_height = inputImage1.getHeight();

vector<vector<RGBQUAD>> image1\_rgbValues;

image1\_rgbValues.resize(image1\_height, vector<RGBQUAD>(image1\_width));

parallel\_for(

blocked\_range2d<int, int>(0, image1\_height, 0, image1\_width),

[&](const blocked\_range2d<int, int>& range) {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

RGBQUAD rgb;

for(int y = y\_start; y < y\_end; y++)

{

for(int x = x\_start; x < x\_end; x++)

{

inputImage1.getPixelColor(x, y, &rgb);

image1\_rgbValues[y][x].rgbRed = rgb.rgbRed;

image1\_rgbValues[y][x].rgbGreen = rgb.rgbGreen;

image1\_rgbValues[y][x].rgbBlue = rgb.rgbBlue;

}

}

}

);

// setup input image 2 array and extract the color data

fipImage inputImage2;

inputImage2.load("../Images/render\_2.png");

int image2\_width = inputImage2.getWidth();

int image2\_height = inputImage2.getHeight();

vector<vector<RGBQUAD>> image2\_rgbValues;

image2\_rgbValues.resize(image2\_height, vector<RGBQUAD>(image2\_width));

parallel\_for(

blocked\_range2d<int, int>(0, image2\_height, 0, image2\_width),

[&](const blocked\_range2d<int, int>& range) {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

RGBQUAD rgb;

for(int y = y\_start; y < y\_end; y++)

{

for(int x = x\_start; x < x\_end; x++)

{

inputImage2.getPixelColor(x, y, &rgb);

image2\_rgbValues[y][x].rgbRed = rgb.rgbRed;

image2\_rgbValues[y][x].rgbGreen = rgb.rgbGreen;

image2\_rgbValues[y][x].rgbBlue = rgb.rgbBlue;

}

}

}

);

//setup output image array

fipImage outputImage = fipImage(FIT\_BITMAP, image1\_width, image1\_height, 24);

int output\_height = image1\_height;

int output\_width = image1\_width;

vector<vector<RGBQUAD>> output\_rgbValues;

output\_rgbValues.resize(output\_height, vector<RGBQUAD>(output\_width));

parallel\_for(

blocked\_range2d<int, int>(0, output\_height, 0, output\_width),

[&](const blocked\_range2d<int, int>& range) {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

for(int y = y\_start; y < y\_end; y++)

{

for(int x = x\_start; x < x\_end; x++)

{

output\_rgbValues[y][x].rgbRed = abs(image1\_rgbValues[y][x].rgbRed - image2\_rgbValues[y][x].rgbRed);

output\_rgbValues[y][x].rgbGreen = abs(image1\_rgbValues[y][x].rgbGreen - image2\_rgbValues[y][x].rgbGreen);

output\_rgbValues[y][x].rgbBlue = abs(image1\_rgbValues[y][x].rgbBlue - image2\_rgbValues[y][x].rgbBlue);

int threshold = 2;

if(output\_rgbValues[y][x].rgbRed > threshold && output\_rgbValues[y][x].rgbGreen > threshold && output\_rgbValues[y][x].rgbBlue > threshold) {

output\_rgbValues[y][x].rgbRed = 255;

output\_rgbValues[y][x].rgbGreen = 255;

output\_rgbValues[y][x].rgbBlue = 255;

}

else{

output\_rgbValues[y][x].rgbRed = 0;

output\_rgbValues[y][x].rgbGreen = 0;

output\_rgbValues[y][x].rgbBlue = 0;

}

outputImage.setPixelColor(x, y, &output\_rgbValues[y][x]);

}

}

}

);

outputImage.save("RGB\_processed.png");

int total = output\_height \* output\_width;

int white\_count = parallel\_reduce(

blocked\_range2d<int,int>(0, output\_height, 0, output\_width),

0,

[&](const blocked\_range2d<int,int>& range, int initValue)->int {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

for(int y = y\_start; y < y\_end; y++)

{

for(int x = x\_start; x < x\_end; x++)

{

if(output\_rgbValues[y][x].rgbRed == 255 && output\_rgbValues[y][x].rgbGreen == 255 && output\_rgbValues[y][x].rgbBlue == 255) {

initValue++;

}

}

}

return initValue;

},

[&](int x, int y)->int{

return x + y;

}

);

cout << "Total Pixels = " << total << endl;

cout << "Total White Pixels = " << white\_count << endl;

double white\_percentage = ((double)white\_count/(double)total)\*100;

cout << "Percentage of White Pixels (%) = " << white\_percentage << "%" << endl;

random\_device rd;

mt19937 mt(rd());

uniform\_int\_distribution<int> randomY(0, output\_height);

uniform\_int\_distribution<int> randomX(0, output\_width);

int redX = randomX(mt);

int redY = randomY(mt);

output\_rgbValues[redY][redX].rgbRed = 255;

output\_rgbValues[redY][redX].rgbGreen = 0;

output\_rgbValues[redY][redX].rgbBlue = 0;

outputImage.setPixelColor(redX, redY, &output\_rgbValues[redY][redX]);

outputImage.save("RGB\_processed.png");

int resultX, resultY;

parallel\_for(

blocked\_range2d<int,int>(0, output\_height, 0, output\_width),

[&](const blocked\_range2d<int, int>& range) {

auto y\_start = range.rows().begin();

auto y\_end = range.rows().end();

auto x\_start = range.cols().begin();

auto x\_end = range.cols().end();

for(int y = y\_start; y < y\_end; y++){

for(int x = x\_start; x < x\_end; x++) {

if(output\_rgbValues[y][x].rgbRed == 255 && output\_rgbValues[y][x].rgbGreen == 0 && output\_rgbValues[redY][redX].rgbBlue == 0) {

if(task::self().cancel\_group\_execution()) {

resultX = x;

resultY = y;

cout << "Red Pixel At X = " << resultX << ", Y = " << resultY << endl;

}

}

}

}

}

);

}

int main() {

cout << "Parallel Run" << endl;

auto t3 = std::chrono::high\_resolution\_clock::now();

parallelRGBProcessing();

auto t4 = std::chrono::high\_resolution\_clock::now();

auto parDuration = std::chrono::duration\_cast<std::chrono::microseconds>(t4 - t3);

cout << "Parallel Time: " << parDuration.count() << " Microseconds" << endl;

return 0;

}