Project 1 report

Sen Wang

03/04/2017

Table of Contents

[**1** **Project summary** 1](#_Toc476395400)

[**2** **Project implementation** 1](#_Toc476395401)

[2.1 Create the project 2](#_Toc476395402)

[2.2 Fill in the serial code 2](#_Toc476395403)

[2.3 Fill in the GPU kernel code 3](#_Toc476395404)

[2.4 Fill in the Main method 3](#_Toc476395405)

[2.5 Fill in the helper function 4](#_Toc476395406)

[**3** **Issues** 5](#_Toc476395407)

[**4** **Results** 5](#_Toc476395408)

[**5** **Conclusions** 8](#_Toc476395409)

1. **Project summary**

Thresholding is a process that takes each pixel value of a picture and changes them to black/white by comparing the value of the pixel with a threshold. Specifically, this project takes a gray sale picture and convert it to a black/white picture using intensity thresholding. As requested, the three images given were thresholded separately through the program, and different threshold values were tried. The thresholded screenshots from CPU and GPU outputs are presented at the end of this report as attachment.

The general structure of the program is described below.

1. Header files to include different libraries;
2. CPU thresholding function to be called by the main method;
3. GPU thresholding function to be called by the main method;
4. Main method;
5. Help functions to perform thresholding parallel with GPU.
6. **Project implementation**

The project is accomplished with C++ language based on Visual Studio 2015, OpenCV 3.2, and CUDA 8.0 template. The implementation of the project are described below in general. Each step will be described in detail following these general description.

1. Create a project based on CUDA 8.0 template in visual studio 2015, load the given kernel file, and setup the OpenCV libraries and workspace etc. ;
2. Fill in the CPU serial code;
3. Fill in the GPU kernel code;
4. Fill in the Main method;
5. Fill in the helper functions;
6. Try different threshold values, and take the screenshots of different images.
   1. Create the project

The project was created with the following steps.

1. Open the visual studio, click new project (“Project 1” in my case), and select CUDA 8.0 runtime.
2. After the project is created, the given kernel file was opened and the content of the given kernel was copied to the newly created project kernel.
3. Next step, Right click the “Project 1”, select add, and add an item. A dummyfile.cpp file was created so that I was able to setup the OpenCV libraries and work space under c++ environment.
4. Right click the “Project 1” , select properties
   1. Under C++/general/Additional Include Directories, copy the full path of “C:\Users\sw5\Desktop\course books\6398\opencv\build\include” where my OpenCV was setup to the Additional Include Directories.
   2. Under Linker/general/Additional Library Directories, copy the full path of

“C:\Users\sw5\Desktop\course books\6398\opencv\build\x64\vc14\lib” where my OpenCV was setup to the Additional library Directories.

* 1. Under Linker/input/Additional Dependencies, write in “opencv\_world320d.lib”.
  2. Under Configuration Properties/Debugging/Working Directory, copy the full path of “C:\Users\sw5\Desktop\course books\6398\opencv\build\x64\vc14\bin” where my OpenCV was setup to the Working Directory.

1. complete
   1. Fill in the serial code

In the CPU serial code, the output image, the input image, and the threshold are used as argument. Involved parameters are rows, cols, index etc. The data at calculated index is compared with the threshold value. If data is less than the threshold value, the output data is assigned with 0. If the data is larger than the threshold value, the output data is assigned with 255.

void cpuThresholding(Mat& output, const Mat& input, unsigned char th)

{

int rows = input.rows;

int cols = input.cols;

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

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

{

int index = j + i\*cols;

if (input.data[index] < th)

output.data[index] = 0;

else

output.data[index] = 255;

}

}

* 1. Fill in the GPU kernel code

The GPU kernel code functions the same as the CPU code (same algorithm) except that the algorithm is executed parallel with GPU. The rows of the image pixels are categorized as blocks, and the columns of the image pixels are categorized as threads. The arguments used here are “dst” as output, “src” as input, and th as threshold. Again, the index is calculated for each pixel and compared with threshold value. 0 and 255 are assigned correspondingly after comparison to each pixel.

cudaError\_t performWithCuda(Mat&, const Mat&, unsigned char th);

// CUDA GPU Kernel

\_\_global\_\_ void gpukernel(unsigned char \*dst, const unsigned char \* src, unsigned char th)

{

// TODO:

// 1- calculate the index of the pointers based on pixel location for each thread

// 2- perform the thresholding

int index = threadIdx.x + blockIdx.x\*blockDim.x;

if (src[index] < th)

dst[index] = 0;

else

dst[index] = 255;

* 1. Fill in the Main method

The main method mainly reads an image with “imread()”, outputs image thresholded by CPU and GPU methods. In addition, if image is loaded successfully, the CPU and GPU thresholding methods will be called to process the image. The processed image will be shown with “imshow()” functions. cvWaitKey(0) was used to pause the image show.

Mat inputImage = imread("C:\\Users\\sw5\\Desktop\\course books\\6398\\Home works\\Project 1\\Desert\_gray.jpg",0);

//1 - 2 - 1 - create an image for the CPU output, and one for the GPU output

Mat cpuThresholdImage(inputImage.rows, inputImage.cols, CV\_8UC1);

Mat gpuThresholdImage(inputImage.rows, inputImage.cols, CV\_8UC1);

// 1-1- if image has no data show an error message

if (!inputImage.data)

printf("Image didn't load properly!\n");

else

{

// 1-2- if image has data

// 1-2-2- call your CPU side code to threshold the image (pass the input image and the cpu output image and the threshold)

cpuThresholding(cpuThresholdImage, inputImage, threshold);

// 1 - 2 - 3 - call the performWithCuda function to create gpu pointers, copy data from host to device, invoke kernel

// and copy results back to host (refer to the above function prototype on line 23 for reference.)

cudaStatus = performWithCuda(gpuThresholdImage, inputImage, threshold);

if (cudaStatus != cudaSuccess)

{

fprintf(stderr, "Cuda failed");

return 1;

}

//1 - 2 - 4 - Use imshow to show the input image, the CPU output and the GPU output.Note: CPU and GPU outputs should look alike.

imshow("Input\_Image", inputImage);

imshow("CPU\_OUTPUT", cpuThresholdImage);

imshow("GPU\_OUTPUT", gpuThresholdImage);

}

* 1. Fill in the helper function

The helper function mainly does the tasks below (unimportant information has been deleted here, please see the kernel.cu for the detailed codes):

1. assign memories for input and output images on the GPU;
2. Transfer information from CPU to GPU
3. Call kernel
4. Transfer processed information from GPU back to CPU.

Arguments used here are “outputImg”, “inputImg”, and “threshhold”. Memories are assigned as “dev\_ptrout” and “dev\_ptrin”.

cudaError\_t performWithCuda(Mat & outputImg, const Mat & inputImg, unsigned char threshold)

{

unsigned char \*dev\_ptrout, \*dev\_ptrin;

}

// allocate input memory

cudaStatus = cudaMalloc((void\*\*)& dev\_ptrin, sizeof(unsigned char)\*inputImg.rows\*inputImg.cols);

// allocate output memory

cudaStatus = cudaMalloc((void\*\*)& dev\_ptrout, sizeof(unsigned char)\*outputImg.rows\*outputImg.cols);

// copy input information from CPU to GPU

cudaStatus = cudaMemcpy(dev\_ptrin, inputImg.data, sizeof(unsigned char)\*inputImg.rows\*inputImg.cols, cudaMemcpyHostToDevice);

// invoke GPU kernel

int grid\_size=inputImg.rows;

int block\_size=inputImg.cols;

gpukernel << <grid\_size, block\_size >> > (dev\_ptrout, dev\_ptrin,threshold);

// check errors during launch of GPU kernel

cudaStatus= cudaGetLastError();

// synchronize

cudaStatus = cudaDeviceSynchronize();

// copy processed information from GPU back to GPU

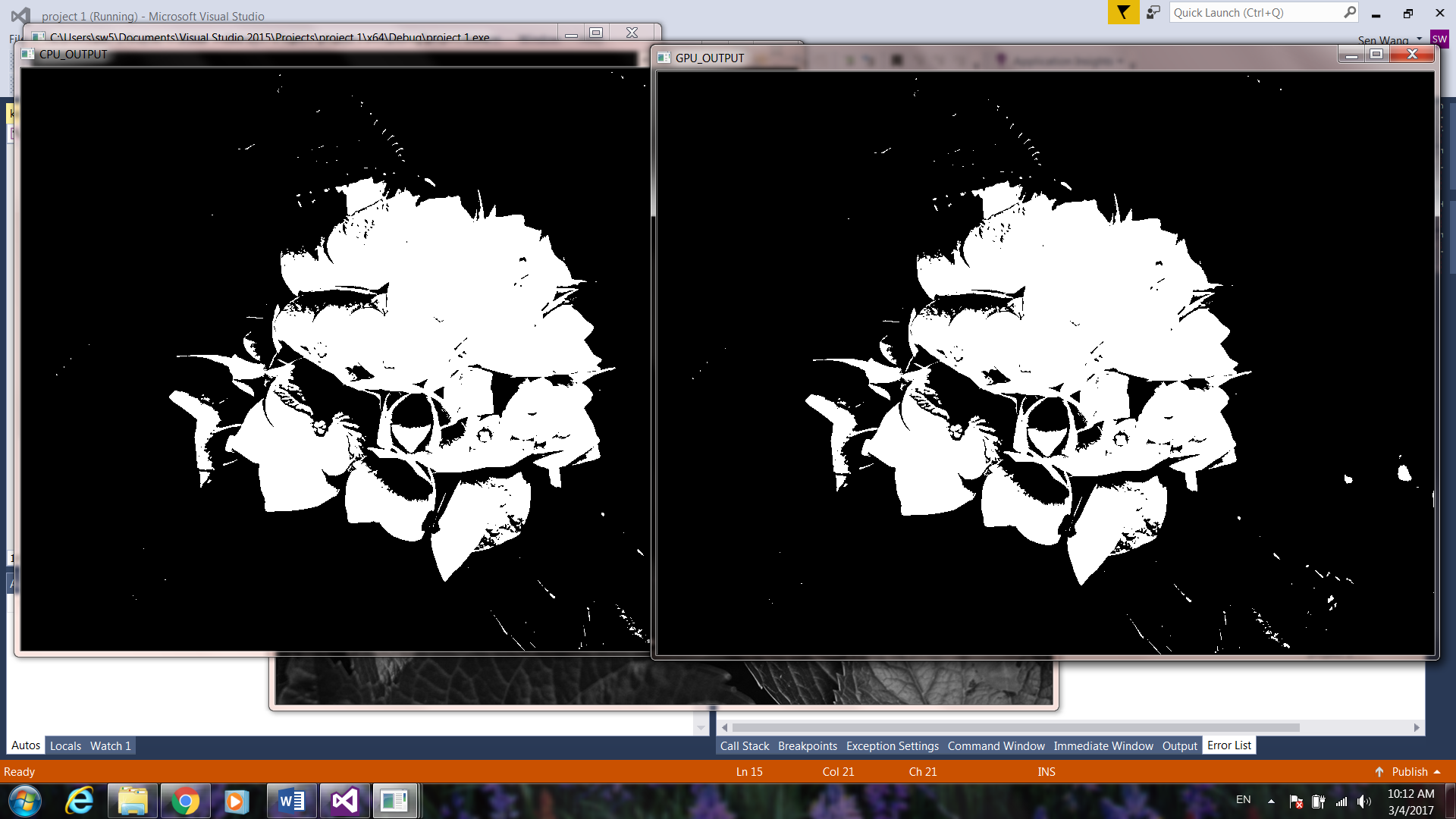
cudaStatus = cudaMemcpy(outputImg.data, dev\_ptrout, sizeof(unsigned char)\*outputImg.rows\*outputImg.cols, cudaMemcpyDeviceToHost);

1. **Issues**

I had an issue with configuring out the right setup for the visual studio i.e. the libraries and linker information, and also I did not realize that there were three places need to be changed to x64 because my computer is a 64 bit device. To solve this problem, I searched a lot on the internet, and fortunately I got the answer before the deadline of the homework.

Another issue I had was to call the CPU and GPU methods in the main method. I had to watch course video about 5 times to completely digest all the information.

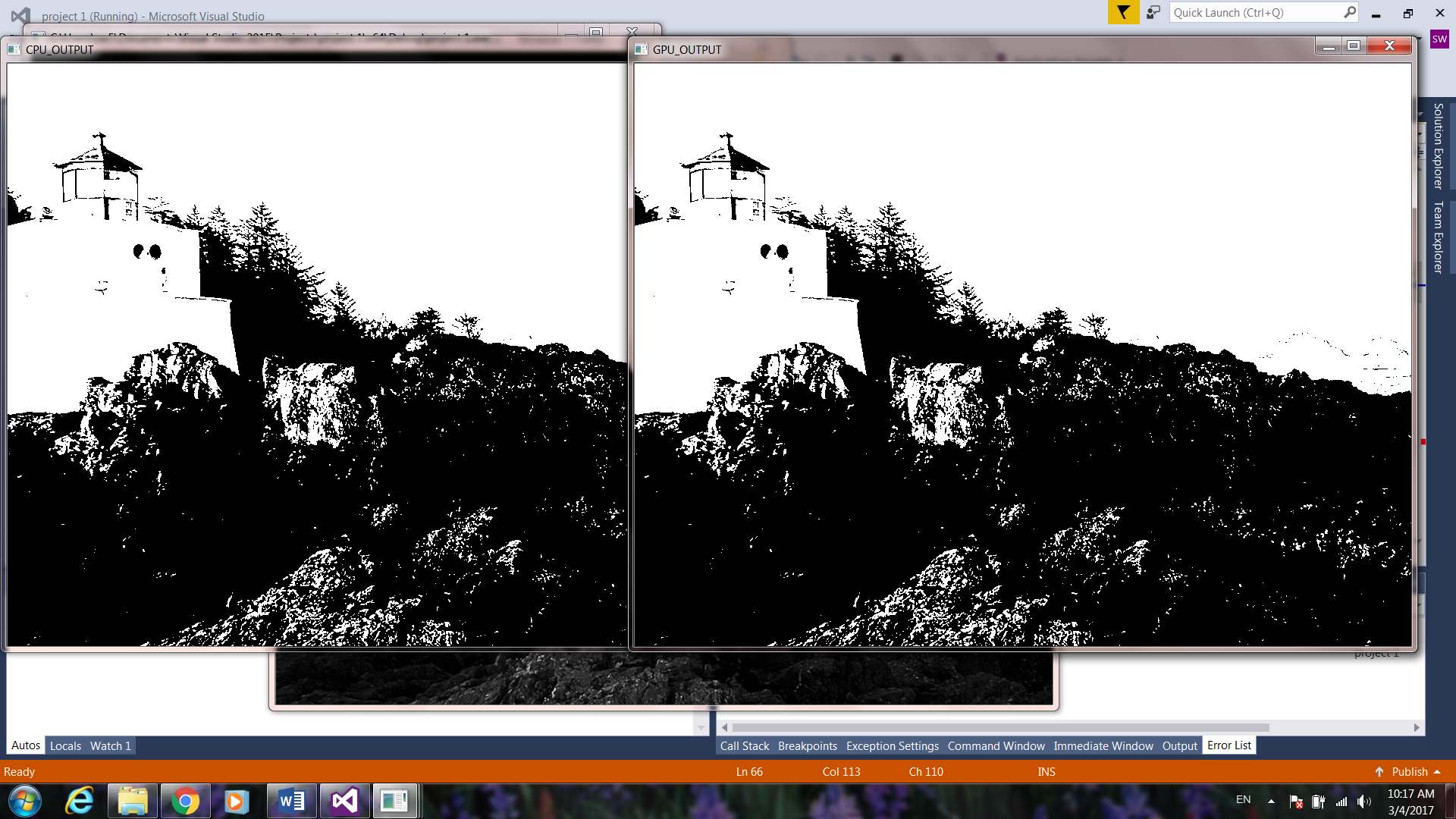
1. **Results**
2. Threshold = 128, Hydrangeas\_gray.jpg



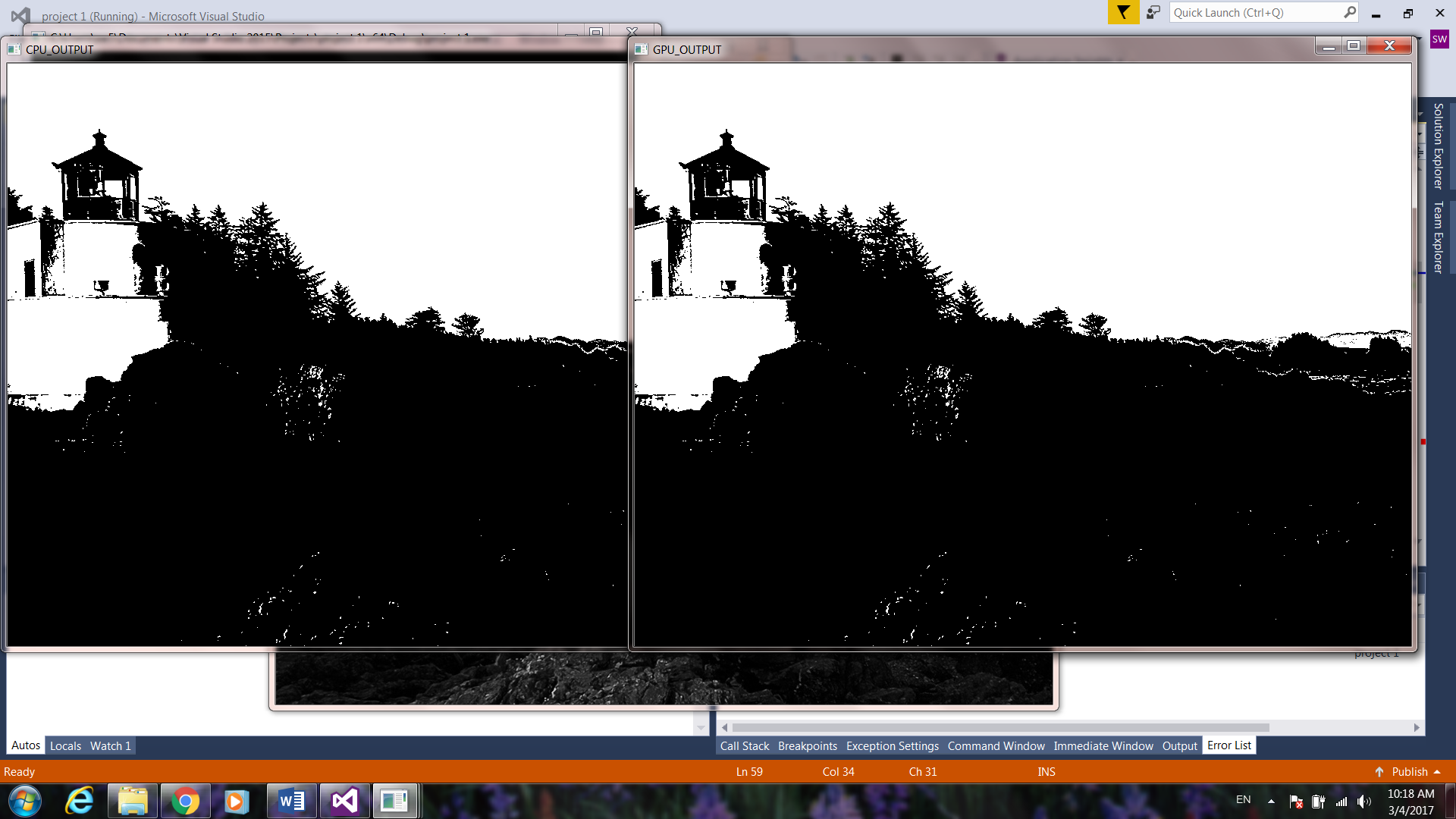
1. Threshold = 50, Hydrangeas\_gray.jpg



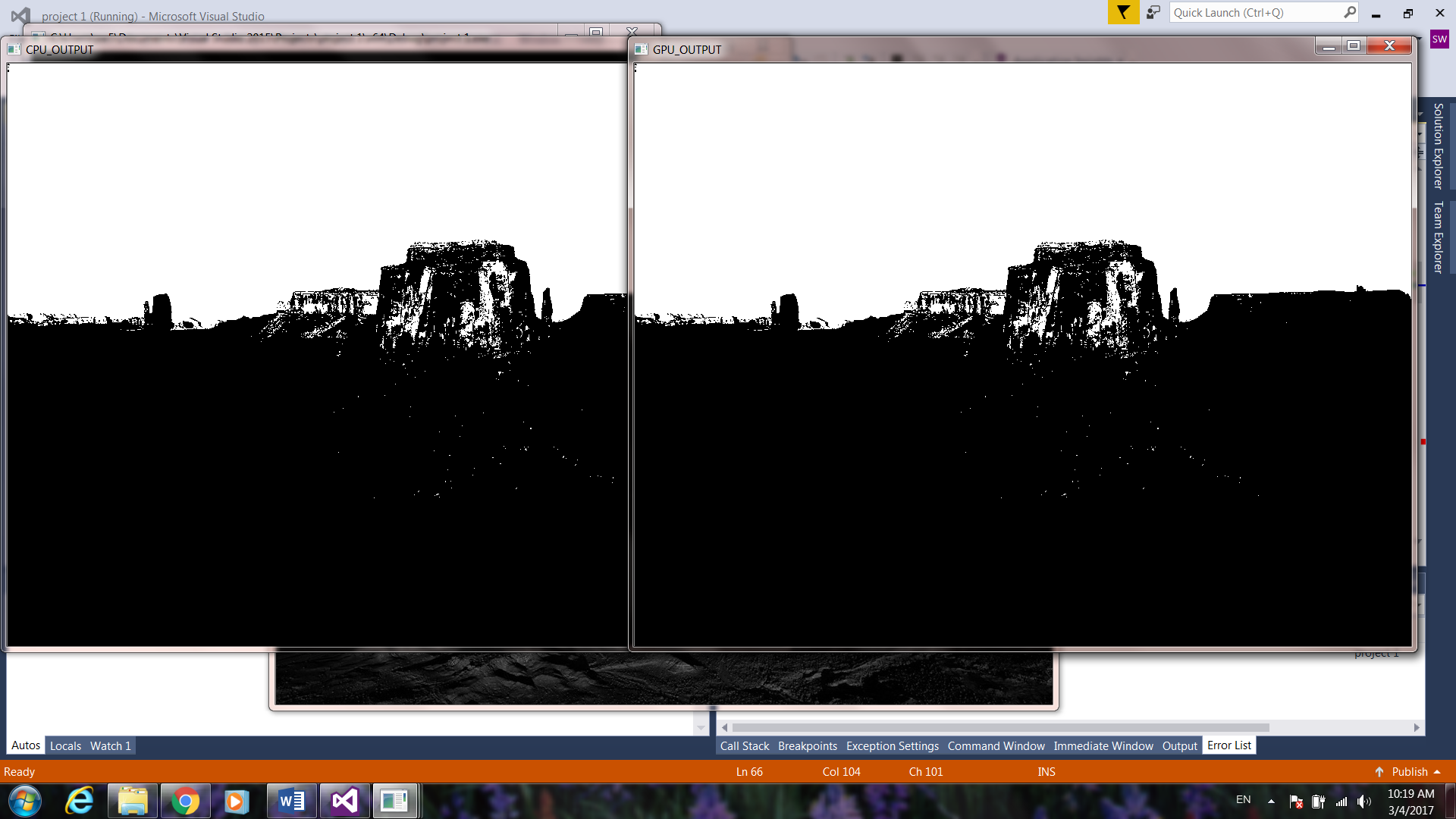
1. Threshold = 50, LightHouse\_gray.jpg



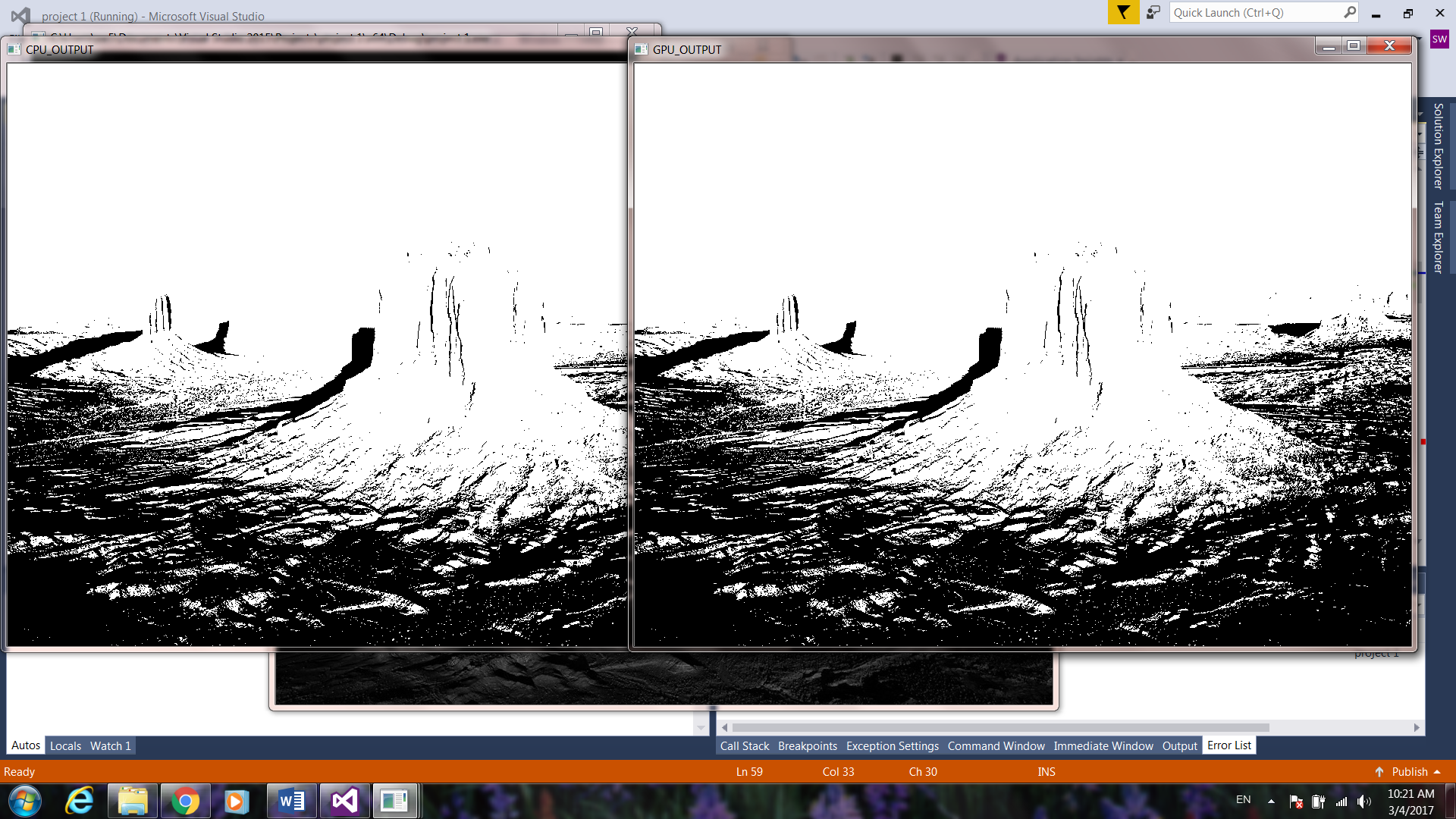
1. Threshold = 128, LightHouse\_gray.jpg



1. Threshold = 128, Desert\_gray.jpg



1. Threshold = 50, Desert\_gray.jpg



1. **Conclusions**

It’s my first time to use visual studio and c++, but I managed to accomplish the project. I learned a lot in terms of both the parallel programming concepts and specific coding through this project. Although the technique required in this project is fairly simple and straightforward, thresholding technique is the base of more complicate image processing process. One real life application I can see with image processing with GPU is auto drive. With auto drive, a live stream of images will be acquired which requires super-fast image processing speed.