Project 2 Report

Sen Wang

04/19/2017

- Write a short section on the project summary.

- Your implementation of both the CPU and GPU code. This section should include any parameters you used on the CPU side code as well as your GPU side code, such as; kernel size, block and grid sizes, conditions, loops, etc.

- Any issues encountered while working on this project, and your attempts to address the issues.

- Results of your program both on the CPU and the GPU.

- Your conclusions about the implementation and the techniques you used and how you see them suitable for a real-life application.

Write a report on your project. The report should be about 3-5 pages long and should contain your own description of both your CPU and GPU side code. The report should contain the following items:

1. **Project Summary**

The project compares three different methods to blur images: CPU method, GPU method with global memory, and GPU method with combined global and shared memory. There are two objectives of this project: (1) learn the different way of implementing the GPU method, i.e. using global/shared memory. Details of implementing the two methods will be discussed below; (2) compare system performance using the three different methods.

The low pass filtering involves two parts: the edges and the other part of the image. The strategy I used to deal with the edge pixels is to pass the input image directly to the output image. The low pass filtering was then used for the rest pixels. However, this strategy only worked for the CPU method apparently. More description on the issues is given below.

In this report, I only included the results from processing one image (the **Desert\_Gray** image). Based on the results, the CPU costs the most computing time, the global +shared memory method costs the second most computing time, and the global memory only method costs the least computing time. However, the GPU methods cost much more overall time than the CPU only method, the longer overall time spent is because of the data transfer between host and device. The GPU method using global/shared memory took 10.17 ms while the GPU using global memory took 6.39 ms. This is because a small part of the image pixels were processed with GPU memory. The 10.17 include both data transfer time from global memory and shared memory.

Table 1 – comparison of the time elapsed of blurring the **Desert\_Gray** image using the three different methods

|  |  |  |
| --- | --- | --- |
| Item | Time elapsed, ms | Overall Time used, ms |
| CPU only | 28 | 28 |
| GPU using global memory | 6.39 | 461 |
| GPU using global/shared memory | 10.17 | 423 |

1. **Codes**

**2.1 CPU code:**

void cpuFilter(Mat& dest, const Mat& src)

{

int rows = src.rows;

int cols = src.cols;

int sum;

// method to deal with edges of the image, bascially use the edge of the original image or the new image.

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

{

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

{

dest.data[j + i\*cols] = src.data[j + i\*cols];

}

}

for (int i = 1; i < rows-1; i++)

{

for (int j = 1; j < cols-1; j++)

{

if ((0 < i < rows - 1) && (0 < j < cols - 1))

{

sum = 0;

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

{

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

{

sum = sum + src.data[(j + x) + (i + y)\*cols];

}

}

int outindex = j + i\*cols;

dest.data[outindex] = sum / ((2 \* W + 1)\*(2 \* W + 1));

}

}

}

}

**2.2 GPU code using global memory:**

1. Kernel

\_\_global\_\_ void gpuKernel(unsigned char \* dst, const unsigned char \*src ,int width, int height )

// this kernel uses the global memory without shared memory, i used width and height as two parameters for juding whether

// the pixels are inside the image size.

{

int index; // pixel location in global memory

//int neighbor;// neighbor location in global memory // i did not use this parameter.

// TODO: define global X and global Y and use these to calculate global offset

// Pixel X and Y in global memory

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

int Y = threadIdx.y + blockIdx.y \* blockDim.y;

// TODO: if global X and global Y are within the image proceed

if (0<=X <width && 0<=Y < height);

// TODO: run the filter 3x3 with two nested for loops

float sum = 0;

index = X + Y\*blockDim.x \* gridDim.x;

for (int j = -W; j <= W; j++) // W is the filter half width

{

for (int i = -W; i <= W; i++) // W is the filter half length

{

// TODO: calculate X and Y for neighboring pixel

// TODO: if neighbor X and Y are not outside of the image boundary do the calculation of neighbor pixel offset

if ((Y + W <= height) && (Y - W >= 0) && (X + W <= width) && (X - W >= 0))

// TODO: calculate the filtered value

{

sum = sum + src[(X + i) + (Y + j)\*blockDim.x \* gridDim.x]; // calculate the sum of 9 pixels

dst[index] = sum / ((2 \* W + 1) \* (2 \* W + 1));

}

else {

dst[index] = src[index]; //I wanted to directly copy the rest of the input image to the output image, so I can get rid of the black line,

// but not successful, I don't how to fix this.

}

}

}

//dst[index] = sum / ((2 \* W + 1) \* (2 \* W + 1)); // calculate average of the 9 values and put into the output image.

}

1. Helper function

cudaError\_t thresholdWithCudaNoShared(Mat & outputImg, const Mat & inputImg)

{

// Allocate GPU buffers for the buffers (one input, one output)

unsigned char \*dev\_dst = 0;

unsigned char \*dev\_src = 0; // these are the gpu side ouput and input pointers

int width = inputImg.size().width;

int height = inputImg.size().height;

cudaError\_t cudaStatus;

cudaEvent\_t start, stop; // These are your start and stop events to calculate

your GPU performance

float time = 0; // This is the gpu time

// Choose which GPU to run on, change this on a multi-GPU system.

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

// TODO: add your code here to allocate the input pointer on the device. Note the size of the pointer in cudaMalloc

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

if (cudaStatus != cudaSuccess)

{

fprintf(stderr, "Cuda failed");

goto Error;

}

// TODO: add your code here to allocate the output pointer on the device. Note the size of the pointer in cudaMalloc

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

if (cudaStatus != cudaSuccess)

{

fprintf(stderr, "Cuda failed");

goto Error;

}

// Copy input data from host memory to GPU buffers.

// TODO: Add your code here. Use cudaMemcpy

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

if (cudaStatus != cudaSuccess)

{

fprintf(stderr, "Cuda failed");

goto Error;

}

// TODO: Launch a kernel on the GPU with one thread for each element. use <<< grid\_size (or number of blocks), block\_size(or number of threads) >>>

dim3 block(K, K, 1);

dim3 grid(inputImg.cols / K, inputImg.rows / K, 1);

// lauch a kernel on the GPU with one thread for each element.

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

gpuKernel <<<grid, block >>> (dev\_dst, dev\_src, width, height);

// TODO: record your stop event on GPU

cudaEventRecord(stop);

// TODO: Synchronize stop event

cudaEventSynchronize(stop);

// TODO: calculate the time ellaped on GPU

cudaEventElapsedTime(&time, start, stop);

printf("Global Memory time=%3.2f ms\n", time);

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

// cudaDeviceSynchronize waits for the kernel to finish, and returns

// any errors encountered during the launch.

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);

goto Error;

}

// TODO: Copy output data from GPU buffer to host memory. use cudaMemcpy

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

if (cudaStatus != cudaSuccess)

{

fprintf(stderr, "Cuda failed");

goto Error;

}

Error:

cudaFree(dev\_src);

cudaFree(dev\_dst);

return cudaStatus;

}

**2.3 GPU code using global / shared memory:**

1. Kernel

\_\_global\_\_ void gpuKernelTiled(unsigned char \* dst, const unsigned char \* src, int width, int height) // This is the shared memory version

{

\_\_shared\_\_ unsigned char Tile[K][K];

// TODO: Declare Tile as shared memory

int lx, ly; // lx and ly are location in shared memory

// TODO: define lx and ly

lx = threadIdx.x;

ly = threadIdx.y;

int X, Y, index = 0;

// X and Y are location of pixel in global memory and index is actual pixel location in global memory

X = threadIdx.x + blockIdx.x \* blockDim.x;

Y = threadIdx.y + blockIdx.y \* blockDim.y;

index = X + Y\*blockDim.x \* gridDim.x;

// TODO: Read from global memory and put in shared memory

Tile[lx][ly] = src[index];

\_\_syncthreads();

// TODO: fill shared memory

float sum = 0; // sum is the filtered value that you will calculate

// TODO: run your for loops for the filtered values

for (int j = -W; j <= W; j++) // W is the filter half width

{

for (int i = -W; i <= W; i++)// W is the filter half length

{

int tmpx, tmpy;

tmpx = lx + i;

tmpy = ly + j;

if ((tmpx>=0)&&(tmpx<K)&&(tmpy>=0)&&(tmpy<K)) // if the pixels are within the block

{

sum += Tile[tmpx][tmpy];

dst[index] = sum / ((2 \* W + 1) \* (2 \* W + 1));

}

else

{

if ((Y + W <= height) && (Y - W >= 0) && (X + W <= width) && (X - W >= 0))

// TODO: calculate the filtered value

{

sum = sum + src[(X + i) + (Y + j)\*blockDim.x \* gridDim.x]; // calculate the sum of 9 pixels

dst[index] = sum / ((2 \* W + 1) \* (2 \* W + 1));

}

else {

dst[index] = src[index]; //I wanted to directly copy the rest of the input image to the output image, so I can get rid of the black line,

// but not successful, I don't how to fix this.

}

}

}

}

//dst[index] = sum / ((2 \* W + 1) \* (2 \* W + 1)); // Here the filtered value will be stored in the output

}

1. Helper function

cudaError\_t thresholdWithCudaWithShared(Mat & destImg, const Mat & srcImg)

{

unsigned char \*dev\_src = 0;

unsigned char \*dev\_dst = 0;

int width = srcImg.size().width;

int height = srcImg.size().height;

cudaError\_t cudaStatus; // cuda status variable for errors on GPU

cudaEvent\_t start, stop; // These are your start and stop events to calculate your GPU performance

float time = 0; // This is the gpu time

// TODO: register your events for GPU

// Choose which GPU to run on, change this on a multi-GPU system.

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

// Allocate GPU buffers for two vectors (One input, one output)

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

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

//target image

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

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

// Copy input vectors from host memory to GPU buffers.

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

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy (CPU ->GPU) failed!");

goto Error;

}

// Launch a kernel on the GPU with one thread for each element.

dim3 block(K, K, 1);

dim3 grid(srcImg.cols / K, srcImg.rows / K, 1);

// TODO: record your start event on GPU

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

gpuKernelTiled <<<grid, block >> >(dev\_dst, dev\_src, width, height);

// invoking the kernel with tiled shared memory

// TODO: record your stop event on GPU

cudaEventRecord(stop);

// TODO: Synchronize stop event

cudaEventSynchronize(stop);

// TODO: calculate the time ellaped on GPU

cudaEventElapsedTime(&time, start, stop);

printf("Shared Memory time=%3.2f ms\n", time);

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

// cudaDeviceSynchronize waits for the kernel to finish, and returns

// any errors encountered during the launch.

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);

goto Error;

}

// Copy output vector from GPU buffer to host memory.

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

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy (GPU -> CPU) failed!");

goto Error;

}

Error:

cudaFree(dev\_src);

cudaFree(dev\_dst);

return cudaStatus;

}

1. **Issues**

The first issue I came cross was how to deal with the four edges of the output image. There was a black line at the edge of the output image without special treatment. For the CPU code, I directly copied the edge pixels of the input image to the output image, and this is successful. However, this strategy did not work for the GPU codes.

The second issue I had is related to using the visual studio. I was trying to debug the problem of black lines on the edge. However, I found that the results were always the same even I completely commented out the GPU global memory kernel. I was not able to figure out the reason for this.

1. **Results**

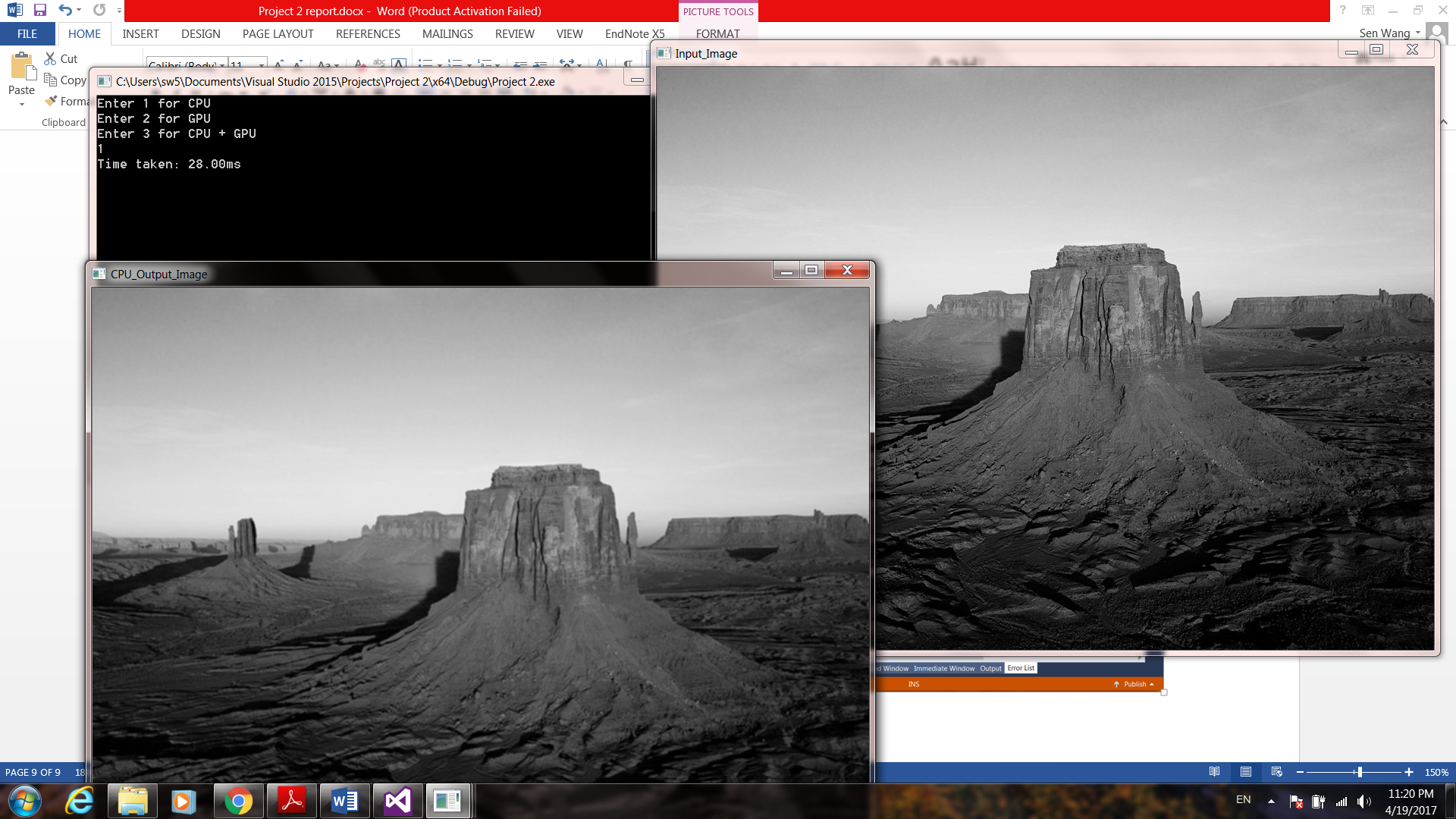


Figure 1 – CPU only method

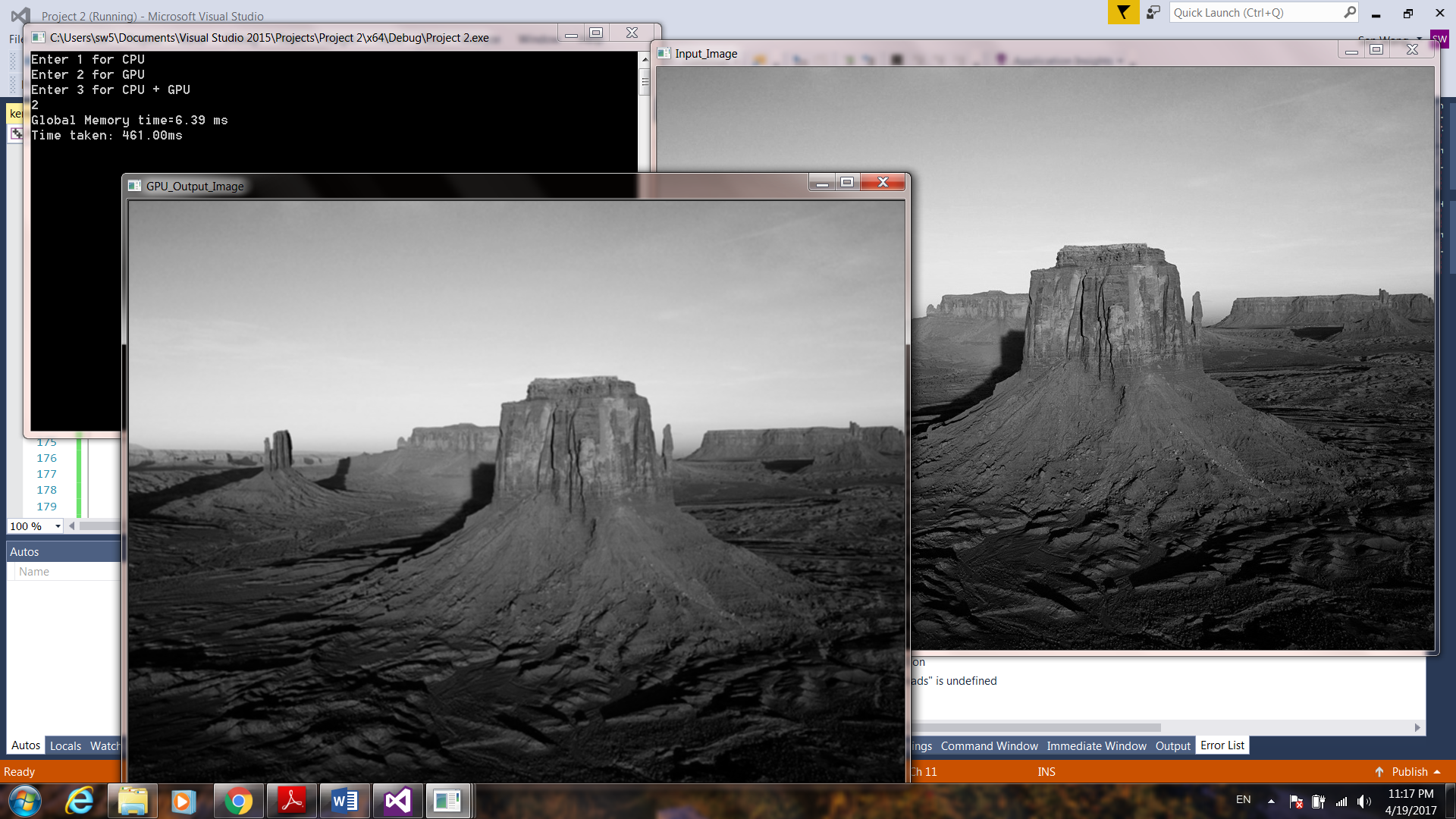


Figure 2 – GPU method using global memory

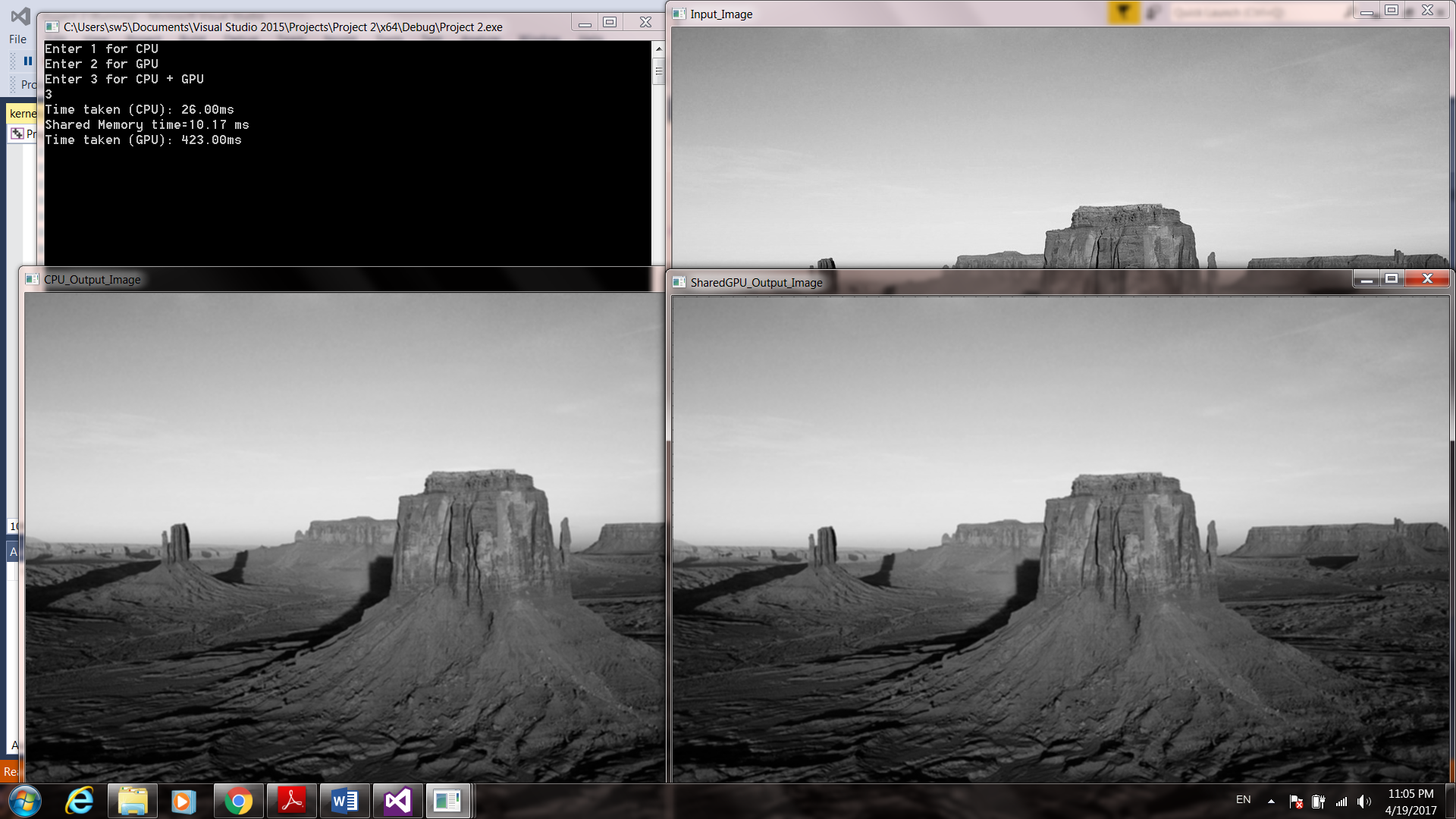


Figure 3 – GPU method using shared/global memory

1. **Your conclusions about the implementation and the techniques you used and how you see them suitable for a real-life application.**

The implementation was successful for the CPU, but not so perfect for the GPU methods because I was not able to fix the edges.

GPU methods are significantly faster than CPU method, the only thing that slows down the overall speed is the data transfer between host and device. GPU method using global memory with proper buffering mechanism can hide the latency on the data transfer. Image processing with shared memory will have a big potential depending on how many times the data in shared memory will be reused. A real time application example of GPU method using global memory is video processing from a drone. For GPU method using shared memory, maybe AutoCAD 3D/2D modeling is a good example, normally only a small part of the model is changed during operation and most part remains the same in AutoCAD modeling, which requires a lot of reuse of the data in shared memory.