**CSC8820**

**Advanced Graphics Algorithms**

Spring 2018

**Project 4**

Due date: 11:59 pm **04/27/2018**

Requirements

In this project you will learn how to write an OpenCL program for simple image processing.

Requirements:

1. Your program should read a color image, convert it to a grayscale image, and save the grayscale image to a file.
   1. The algorithms for converting color to grayscale can be found [here](http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/) or [here](http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/).

I have used this algorithm (<http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/)> for converting color to grayscale image. The attached files can read a color image, convert it to a grayscale image, and save the grayscale image to a file.

* 1. You must use one of the two image loading libraries: [CImg](https://github.com/dtschump/CImg) or [FreeImage](http://freeimage.sourceforge.net/). This will make it easier for my TA and me to build your program.

Since I built in Mac, I have used FreeImage for loading image (http://freeimage.sourceforge.net/requirements.html).

1. Implement both a parallel version of this algorithm in C/C++ and OpenCL, and also a sequential version in C or C++.
   1. Compare the performance of the parallel version and the sequential version. Is the parallel version faster than the sequential version?

The 4 implementations were run 10 times in a row with file names, c++\_Parallel, c++\_Sequential, opencl\_Parallel and opencl\_Sequential.cpp. The output is as follows:

OpenCL parallel code ran in: 1.689416 msecs.

OpenCL sequential code ran in: 2.689416 msecs.

C++ parallel code ran in: 1.999916 msecs.

C++ sequential code ran in: 3.641999 msecs.

By executing the programs, OpenCL implementation took in average ~1.6 ms, the C++ implementation took in average ~1.9 ms on parallel processing which was faster than sequential.

* 1. Vary the number of work items. Does it make a difference in performance (i.e. time)?

Yes, multiple image processing did increased the processing time for all of them but followed the same order of time magnitude for conversions.

* 1. Compare the performance of the parallel version of your algorithm on CPU and GPU. Which one is faster?

I compared the parallel version of the algorithm on CPU and GPU and all the codes on GPU ran faster than on CPU, following were execution timings:

OpenCL parallel code ran in: 1.500416 msecs.

OpenCL sequential code ran in: 1.277416 msecs.

C++ parallel code ran in: 1.788816 msecs.

C++ sequential code ran in: 2.772999 msecs.

* 1. Include a brief report to describe your experiments and results.

Method:

4 programs were written to compare OpenCL and C++ color to gray scale image conversion and ran on parallel and sequential flows and on CPU and GPU’s respectively. Following is pseudocode for C++ program:

*void processUsingCpu(std::string input\_file, std::string output\_file) {*

*// pointers to images in CPU's memory (h\_) and GPU's memory (d\_)*

*//load the image and give us our input and output pointers*

*preProcess(&h\_rgbaImage, &h\_greyImage, &d\_rgbaImage, &d\_greyImage, input\_file);*

*rgbaToGreyscaleCpu(h\_rgbaImage, h\_greyImage, numRows(), numCols());*

*if (err < 0) {*

*//Couldn't print!*

*std::cerr << "Couldn't print timing information! STDOUT Closed!" << std::endl;*

*exit(1);*

*}*

*//check results and output the grey image*

*postProcess(output\_file, h\_greyImage);*

*}*

*// Serial implementation for running on CPU using a single thread.*

*Following is pseudocode for OpenCL program:*

*void processUsingCuda(std::string input\_file, std::string output\_file) {*

*// pointers to images in CPU's memory (h\_) and GPU's memory (d\_)*

*//load the image and give us our input and output pointers*

*preProcess(&h\_rgbaImage, &h\_greyImage, &d\_rgbaImage, &d\_greyImage, input\_file);*

*size\_t numPixels = numRows()\*numCols();*

*checkCudaErrors(cudaMemcpy(h\_greyImage, d\_greyImage, sizeof(unsigned char) \* numPixels, cudaMemcpyDeviceToHost));*

*//check results and output the grey image*

*postProcess(output\_file, h\_greyImage);*

*}*

*// kernel which is run in parallel by many GPU threads.*

*\_\_global\_\_*

*void rgbaToGreyscaleCudaKernel(const uchar4\* const rgbaImage,*

*unsigned char\* const greyImage,*

*const int numRows, const int numCols)*

*{*

*//First create a mapping from the 2D block and grid locations*

*//to an absolute 2D location in the image, then use that to*

*//calculate a 1D offset*

*const long pointIndex = threadIdx.x + blockDim.x\*blockIdx.x;*

*if(pointIndex<numRows\*numCols) {*

*// this is necessary only if too many threads are started*

*uchar4 const imagePoint = rgbaImage[pointIndex];*

*greyImage[pointIndex] = .299f\*imagePoint.x + .587f\*imagePoint.y + .114f\*imagePoint.z;*

*}*

*}*

*// Parallel or sequential implementation for running on GPU using multiple threads.*

Results and Conclusions:

By executing the programs, OpenCL implementation took in average ~1.6 ms, the C++ implementation took in average ~1.9 ms on parallel processing which was faster than sequential. Multiple image processing did increase the processing time for all of them but followed the same order of time magnitude for conversions. Finally, the comparison of the parallel version of the algorithm on CPU and GPU and all the codes on GPU ran faster than on CPU.

1. Upload your source code and report in a ZIP file to iCollege under the folder Project 4. Do not submit via email.

Uploaded a zip folder with these.