Assignment № 1

Deadline: 11:55pm Oct. 11, 2018

Problem 1: Julia Set (40%)

For the first problem in your first assignment, let's write an exciting program named Julia Set.

Learning Outcomes

- Be able to compile and execute GPU programs
- Be able to do the error check
- Be able to use "correct" indexing method

You need to write a code to draw slices of the Julia Set, as shown in figure 1. For the uninitiated, the Julia Set is the boundary of a certain class of functions over complex numbers. Undoubtedly, this sounds not that interesting. However, for almost all values of the function's parameters, this boundary forms a fractal, one of the most exciting and beautiful curiosities of mathematics.

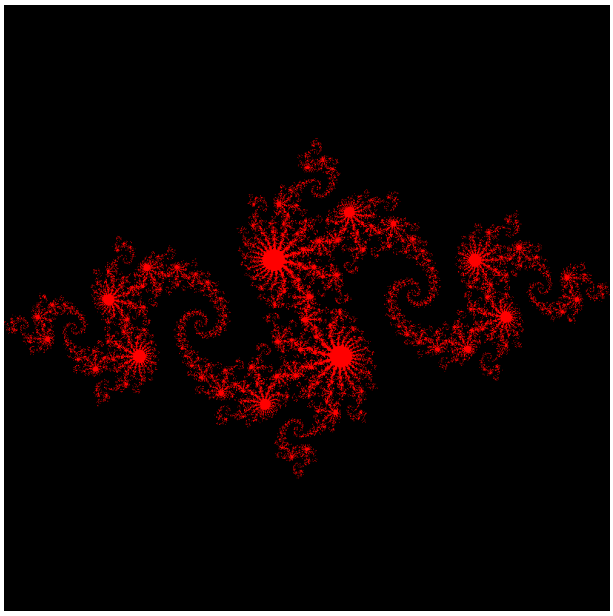

Figure 1: The figure of Julia Set you are going to generate.

The calculations involved in generating such a set are quite simple. At its heart, the Julia Set evaluates a simple iterative equation for points in the complex plane. A point is not in the set if the process of iterating the equation diverges for that point. That is, if the sequence of values produced by iterating the equation grows toward infinity, a point is considered outside

the set. Conversely, if the values taken by the equation remain bounded, the point is in the set. Computationally, the iterative equation in question is remarkably simple: $Z_{n+1} = Z_n^2 + C$. Computing an iteration of this equation would therefore involve squaring the current value and adding a constant to get the next value of the equation.

As this is not a math course, we have already implemented a CPU version Julia Set Generator in **julia.cu**. All you need to do is rewriting the CPU code to a GPU one, and hopefully optimize the code to beat others.

1. generate the Julia Set with GPU. (4%)
2. in the tutorial, we mentioned block and thread. In this problem, you need to try 5 different blocks dims and threads dims for 1D indexing as well as 5 different blocks dims and threads dims for 2D indexing. Try extreme cases to get different execution time. You need to cover these six combinations: $<<<(DIM, DIM), (1, 1)>>>$, $<<<(?, ?), (32, 32)>>>$, $<<<(?, ?), (3, 3)>>>$ for 2D, and $<<<DIM * DIM, 1>>>$, $<<<?, 1024>>>$, $<<<?, 3>>>$ for 1D(16%)
3. write a report (two or three paragraphs) to compare the execution time and the readability of different indexing methods. (8%)
4. in this report, you also need to compare the execution time between CPU (on Azure VMs) and GPU. (4%)
5. code style, error check, memory free etc. (8%)

Problem 2: Gaussian Blur (60%)

In this problem, you are going to blur an image (a poster of iron man, if you do not like him, feel free to use others, as long as it is large enough and in ppm format).

Learning Outcomes

- Understand CUDA variable types
- Be able to choose "best" variable types

To do this, imagine that we have a filter, which is a square array of weight values. For each pixel in the image, imagine that we overlay this square array of weights on top of the image such that the center of the weight array is aligned with the current pixel. To compute a blurred pixel value, we multiply each pair of numbers that line up. In other words, we multiply each weight with the pixel underneath it. Finally, we add up all of the multiplied numbers and assign that value to our output for the current pixel. We repeat this process for all the pixels in the image.

Noticed that the input is a color image that has three channels. The image is coded as RGBARGBARGBARGBA..., the first element is the red value of the first pixel, and the second element is the green value of the first pixel, so on so forth. You need to blur the three channels separately.

You must fill in the **your_gaussian_blur** function in **blur.cu** to perform the blurring of the **PPMImage**, using the array of weights shown in the example, and put the result in the **PPMImage**. After executing the **your_gaussian_blur** function, you need to save the image for grading.

Here is an example of computing a blur, using a weighted average, for a single pixel in a small image.

Array of weights:

0.05	0.1	0.05
0.1	0.4	0.1
0.05	0.1	0.05

Image:

1	2	5	2	0	3
3	2	5	1	6	0
4	3	6	2	1	4
0	4	0	3	4	2
9	6	5	0	3	9

The red 6 will be replaced by $2 * 0.05 + 5 * 0.1 + 6 * 0.05 + 3 * 0.1 + 6 * 0.4 + 2 * 0,1 + 4 * 0.05 + 0 * 0.1 + 3 * 0.05$.

1. generate the blur image. (6%)
2. we have seen the differences between register, constant memory, shared memory and global memory in the tutorial. In this problem, you need to declare the filter as localVar, ConstantVar and GlobalVar, meanwhile declare the input image as sharedVar, GlobalVar to compare the performance. In total, you need to try 6 different combinations (18%) (You need to submit 6 CUDA files)
3. For each case in 2, you are supposed to measure the execution time 10 times and choose the fast one as the result. Write a report to compare and **explain** the experiment results. Also, try to explain why we choose the fast one rather than the average. (27%)
4. code style, error check, memory free etc. (9%)

Reference

https://developer.nvidia.com/cuda-example

https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-separable-convolution

