

ECSE 420 – Parallel Computing – Fall 2016

Lab 1 – Simple CUDA Processing

Due: October 11, 2016 at 11:59 pm

In this lab, you will write code for simple signal processing, parallelize it using CUDA, and write a report summarizing your experimental results. We will use PNG images as the test signals.


Each group should submit a single zip file with the filename Group <your group number> Lab 1.zip.

1. Image Rectification

Rectification produces an output image by repeating the following operation on each element of an input image:

$$output[i][j] = \begin{cases} input[i][j] & \text{if } input[i][j] \geq 0 \\ 0 & \text{if } input[i][j] < 0 \end{cases}$$

Figure 1 shows rectification performed on a small array.



-4	0	4	3
2	2	-1	-5
4	-1	0	0
4	4	1	2

0	0	4	3
2	2	0	0
4	0	0	0
4	4	1	2

Figure 1: Rectification Example

Figure 1: Rectification

Of course, since pixel values are already in the range [0,255], rectifying them will not change their values, so you should “center” the image by subtracting 127 from each pixel, then rectify, then add back 127 to each pixel (or equivalently, compare the pixel values to 127 and set equal to 127 if lower).

Write code that performs rectification. Parallelize the computation using CUDA kernels. Measure the runtime when the number of threads used is equal to $\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$, and plot the speedup as a function of the number of threads. Discuss your parallelization scheme and your speedup plots and make reference to the architecture on which you run your experiments. Include in your discussion an image of your choice (not the provided test image) and the output of rectifying that image.

To check that your code runs properly, the TA will run the following command:

```
./rectify <name of input png> <name of output png> < # threads>
```

When the input test image is “test.png”, the output of your code should be identical to “test_rectify.png”. You can use the “test_equality” code provided by TAs to check if two images are identical. The grader should be able to run your code with different numbers of threads, and the output of your code should be correct for any of those thread counts.

2. Pooling

Pooling compresses an image by performing some operation on regularly spaced sections of the image. For example, Figure 2 shows max-pooling on 2×2 squares in an image. In this case, the operation is taking the maximum value in the section, and the sections are the disjoint 2×2 squares.

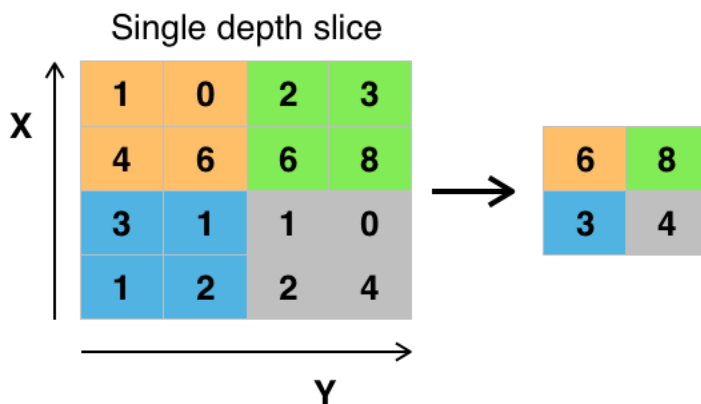


Figure 2: Max Pooling

Write code which performs 2×2 max-pooling. Analyze, discuss, and show an example

image as described in the first section.

The grader will run the following command:

```
./pool <name of input png> <name of output png> < # threads>
```

When the input test image is “test.png”, the output of your code should be identical to “test pool.png”. Note that if the input image is of size m by n , then the output of 2×2 max- pooling will be of size $m/2$ by $n/2$. Do not worry about the corner case in which the image cannot be divided perfectly into 2×2 squares– you may assume that the input test image will have equal width and height.