Lab 1

ECSE 420 – Parallel Computing – Fall 2016

Released: September 30, 2016
Due: October 14, 2016 at 11:59 pm

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

Submit a single zip file with the filename Group_<your group number>_Lab_1.zip.

1. **Rectification**
   Rectification produces an output image by repeating the following operation on each element of an input image[1]:

$$output[i][j] = \begin{cases} input[i][j] & \text{if } input[i][j] \geq 0 \\ 0 & \text{if } input[i][j] < 0 \end{cases} \quad (1)$$

   Figure 1 shows rectification performed on a small array.

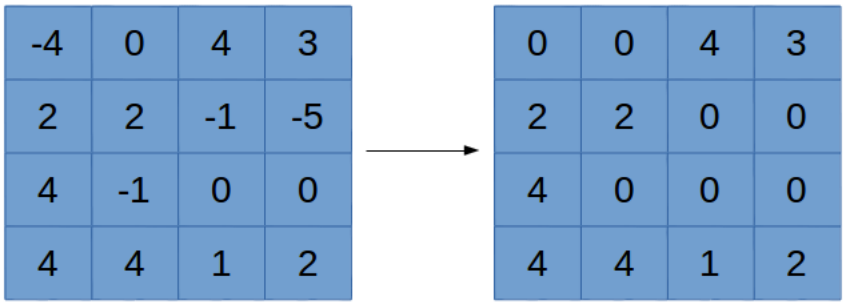

Figure 1: Rectification

   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 which performs rectification. Parallelize the computation using multiple threads. You may use either Pthreads or OpenMP. Measure the runtime when the number of threads used is equal to $\{1, 2, 4, 8, 16, 32\}$, 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 of the computer 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 grader 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 to check if two images are identical by running "gcc test_equality.c lodepng.c" and then "./a.out <image 1> <image 2>". The grader should be able to run your code with 1, 2, 4, 8, 16, or 32 threads, and the output of your code should be correct for any of those thread counts.

[1]Rectification is called an "embarrassingly parallel" algorithm because each operation is independent of every other operation, so the computation as a whole is completely parallelizable. For such an algorithm, the amount of speedup you can achieve is limited only by the amount of hardware available and the amount of OS overhead required to make use of all the hardware.

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 2x2 squares in an image. In this case, the operation is taking the maximum value in the section, and the sections are the disjoint 2x2 squares.

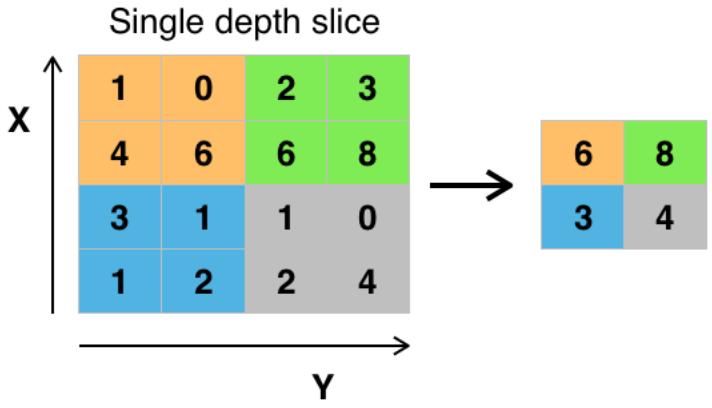

Figure 2: Max pooling

Write code which performs 2x2 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 2x2 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 2x2 squares– you may assume that the input test image will have even width and height.

3. **Convolution**

Convolution is a slightly more complicated operation than rectification and pooling, but it is still highly parallelizable. For each pixel in the input image, convolution computes the corresponding pixel in the output image using a weighted sum of the input pixel and its neighbors. That is:

$$output[i][j] = \sum_{ii=0}^{2} \sum_{jj=0}^{2} input[i+ii-1][j+jj-1]w[ii][jj],\ 1 \leq i \leq m-1,\ 1 \leq i \leq n-1 \quad (2)$$

where m is the number of rows in the input image, and n is the number of columns in the input image. Since we are using 3x3 weight matrices and the "valid padding" definition of convolution ($1 \leq i \leq m-1$, $1 \leq i \leq n-1$), the output image will be of size $m-2$ by $n-2$. Figure 3 illustrates the convolution operation for a certain weight matrix, W.

Write code which performs convolution. For this lab, you only need to implement convolution using 3x3 weight matrices. Before you convert the output to unsigned chars and save the file, you should clamp the output of the convolution between 0 and 255 (i.e., if a value in the output is less than 0, you should set it equal to 0, and if a value is greater than 255, you should set it equal to 255). The header file "wm.h" contains the weight matrix you should use. Analyze, discuss, and show an example image as described in the first section.

The grader will run the following command:

./convolve <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_convolve.png".

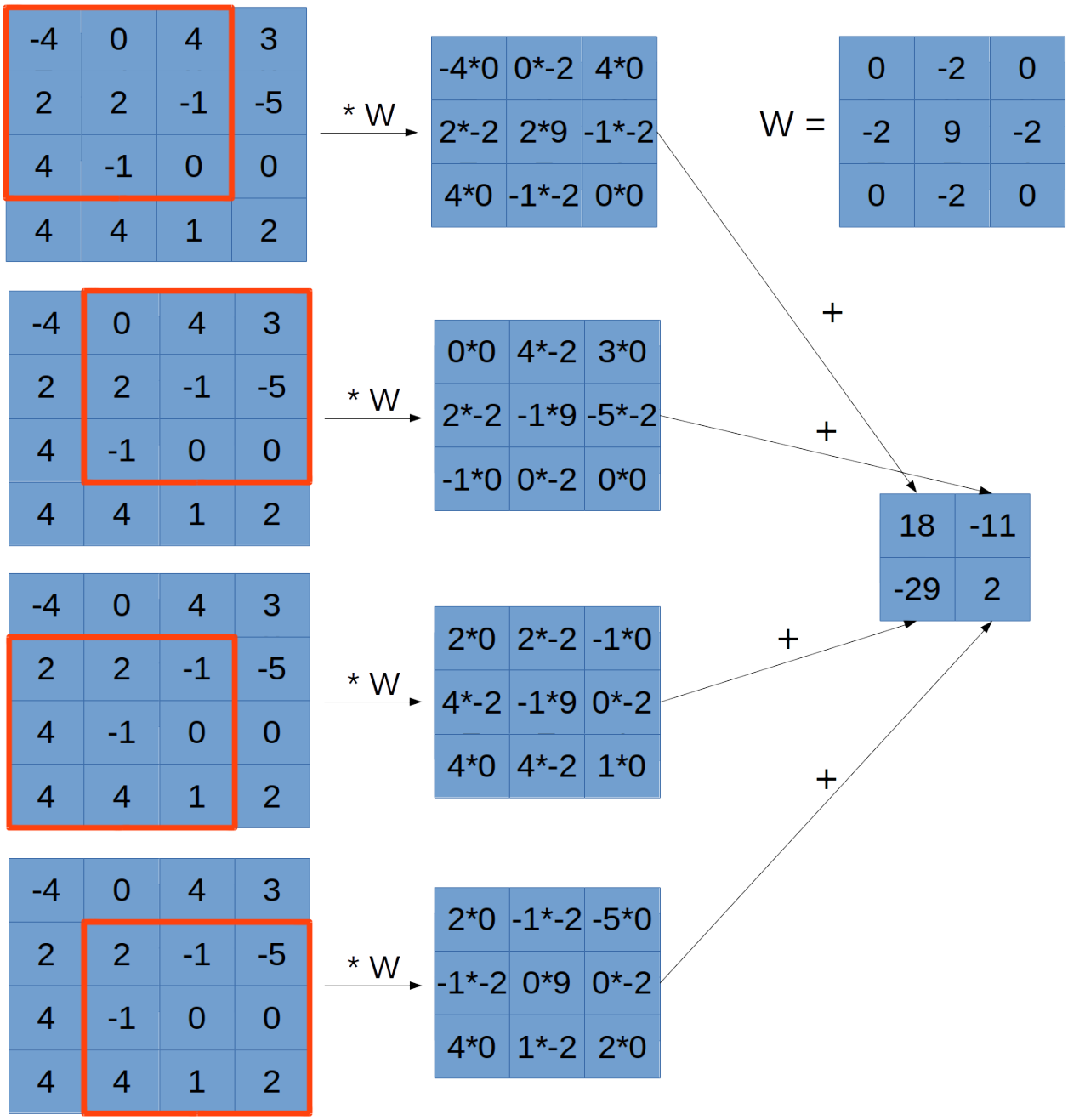

Figure 3: Convolution

Submission Instructions

Your submitted zip file should contain the following:

- a PDF report
- a Makefile
- whatever source files you need such that the Makefile produces the correct executables

To test your submission, the grader will unzip your file, enter the unzipped directory, run "Make", and then run the commands listed in the sections above. Your code must compile and run on the Trottier Linux machines.

The grading scheme is as follows:

- Rectification - 20%
- Pooling - 20%
- Convolution - 20%
- Report - 40%