Box Blur Filter

Prof. Naga Kandasamy
ECE Department, Drexel University

This assignment, worth ten points, is due May 31, 2023, by 11:59 pm. You can work on it in a group of up to two people. Please submit original code.

Implement a parallel version of a box blur filter, suitable for execution on the Graphics Processing Unit (GPU). A *box blur* is a linear filter in the spatial domain in which each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image. Therefore,

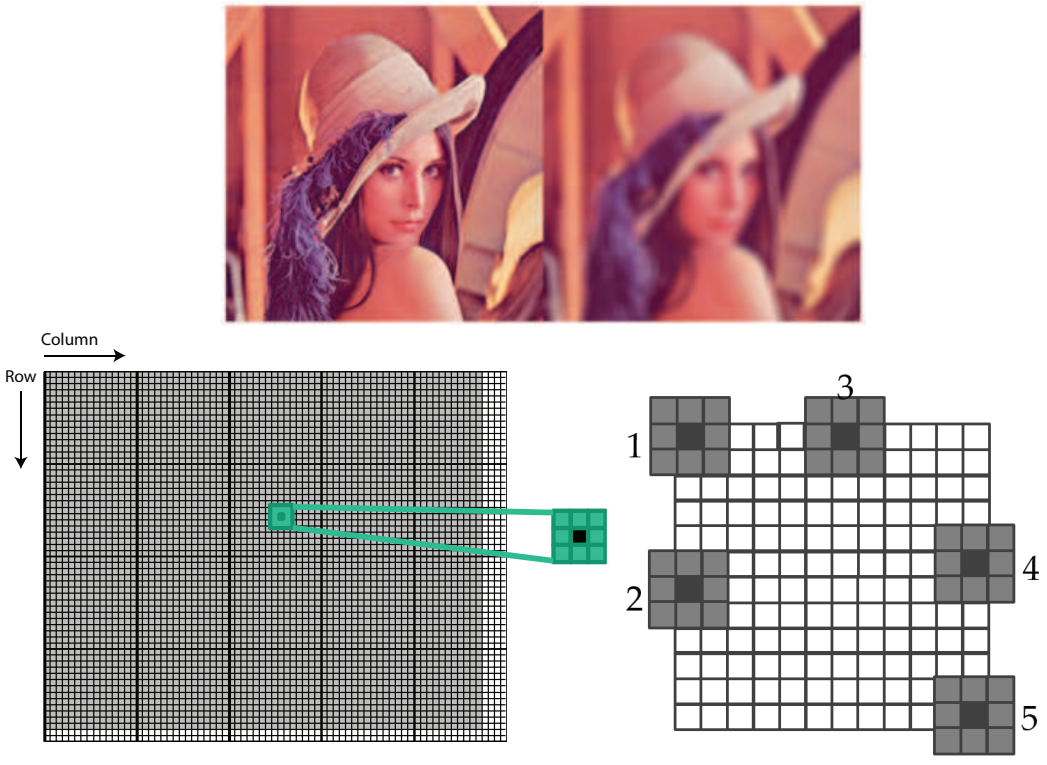

Figure 1: Illustration of the blurring process along with some examples of boundary cases. From: David Kirk and Wen-mei Hwu.

it is a form of low-pass or "blurring" filter. In matrix form, a 3×3 box blur can be written as

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Figure 1 illustrates the blurring process. For an interior pixel, the blur value is simply the average of nine values: the eight surrounding pixels as well as itself. For pixels residing along the boundary, for example, cases 1 and 5, the average is calculated over four values, and in cases 2, 3, and 4, the average is calculated over six values. The snippet of the serial implementation follows.

```
1  /* The input image is called in; the coutput image is called out. */
2
3  #define BLUR_SIZE 1
4
5  int pix, i, j;
6  int row, col, curr_row, curr_col, num_neighbors;
7  float blur_value;
8
9  /* Iterate over each pixel in size x size image. */
10 for (pix = 0; pix < size * size; pix++) {
11     row = pix/size; /* Row number of pixel. */
12     col = pix % size; /* Column number of pixel. */
13
14     /* Apply box filter to the current pixel. */
15     blur_value = 0.0;
16     num_neighbors = 0;
17     for (i = -BLUR_SIZE; i < (BLUR_SIZE + 1); i++)
18         for (j = -BLUR_SIZE; j < (BLUR_SIZE + 1); j++) {
19             curr_row = row + i;
20             curr_col = col + j;
21             /* Check for boundary conditions. */
22             if ((curr_row > -1) && (curr_row < size) &&\
23                 (curr_col > -1) && (curr_col < size)) {
24                     blur_value += in[curr_row * size + curr_col];
25                     num_neighbors += 1;
26             }
27         }
28
29     /* Write the averaged blurred value out. */
30     out[pix] = blur_value/num_neighbors;
31 }
```

The `for` loop in line 10 iterates over all pixels of the `size` × `size` image. It obtains the `(row, col)` position of that pixel within the image and calculates its blur value by averaging the values of the neighboring pixels. Line 22 checks for the various boundary cases discussed earlier.

Using the provided sequential program as a starting point, develop a parallel version of the box blur filter using CUDA. The program accepts image height as the argument. Running it as

```
$ ./blur_filter 512
```

will create a 512×512 image and populate it with random floating-point values. The solution provided by your GPU implementation is compared to that generated by the reference code. Complete the *compute_on_device()* and *blur_filter_kernel()* functions, and upload all source files needed to run your code as a single zip file on BBLearn. The code must compile and run on the NVidia 1080 GTX GPU on the `xunil-05` machine.[1] Also, include a brief report describing: (1) the design of your GPU implementation, using code or pseudocode to clarify the discussion; (2) the speedup obtained over the serial version for image sizes of 4096×4096 and 8192×8192 elements. Ignore the overhead due to CPU/GPU communication when reporting speedup.

Implementation Note: For this assignment, you may simply use GPU global memory to store the various data structures used by your kernel. Use of shared memory is not necessary.

Referring back to the serial code, we are parallelizing the outer *for* loop that iterates over each pixel in the image (lines 10 through 31). Each pixel can be blurred in parallel by a GPU thread as long as the thread identifies the row and column location of the pixel of interest.

[1] Run `make clean` in your source directory. We must be able to build your code from source and don't want pre-compiled executables or intermediate object files.