In this lab, you will be implementing some simple image processing commands in CUDA. Begin by opening the Colab notebook:

`https://github.com/TeachingUndergradsCHC/modules/blob/master/Programming/cuda/cudaBlur.ipynb`

Follow the directions in that notebook to set up CUDA in your Colab instance. One of the directions asks you to upload the files that we use this lab. The files are available from the module repo:

`https://github.com/TeachingUndergradsCHC/modules/blob/master/Programming/cuda/`

These files are the library code that manages images in the .ppm file format (`ppmFile.h` and `ppmFile.c`) and also the image that you want to process; the sample I provide is `640x426.ppm`.

The first really interesting code cell in the notebook is the second to last one. It runs the "Hello World" program discussed in class. Run this example to check that everything is working; it should print out a series of messages, one per thread.

Spend a bit of time playing with the program to better understand how it works. In particular, I suggest that you change the numbers in the `<<< >>>` in `main`. These control the numbers of blocks and threads per block respectively.

Once you're comfortable with the "Hello World" program, it's time to begin the main part of lab. The last code cell in the notebook is the first version of our image processing program. This version removes all the red from our sample image (`640x426.ppm`) and creates a new file `out.ppm`. (You may have to refresh the file panel to the left by hitting the button with circular arrow to see `out.ppm`.) Download this file and observe that it looks basically the same as our original image, but the colors are slightly shifted. The image is stored as an array of RGB values, each color in a separate channel of the pixel.

Let's look through the kernel to see how it works; it's the function `kernel`. This kernel is called once for each pixel of the input image. First, the variables `i` and `j` are set to the x and y coordinates of the pixel for which this call is responsible. The kernel calls are groups into a 2D grid of blocks and each block is itself 2 dimensional. The values of `blockIdx.x` and `blockIdx.y` give the coordinates of the block within the 2D grid of blocks. Each block has dimensions `blockDim.x` by `blockDim.y`. The values of `threadIdx.x` and `threadIdx.y` are the coordinates of the thread within its block.

The image itself is stored linearly as triples of integers for each pixel. The three integers representing a pixel are called the *red channel*, the *green channel*, and the *blue channel* respectively because they store the amount of red, green, and blue in their pixel. The given code simply changes the value in the red channel to 0, leaving the other channels unchanged. This has the effect of removing all the red from the image.

As a first task, let's change this kernel to convert the image into grayscale (black and white). To do this, take the values of the red, green, and blue channels and average them (add them up and divide by 3). Set the value of all three channels to be this average. Run the resulting program and verify that `out.ppm` is now a grayscale image before proceeding.

Next, you will modify the kernet to create a blur effect. This is also done by taking an average, but a different sort of average. Each channel is blurred separately— each gets the average of that channel's value for nearby pixels. Change the kernel to take an integer argument `k` and have `main` call it with value 5. The kernel should use this argument as the distance over which to compute the average. Specifically, the red value of a pixel should be set to the average of the original red values of all the pixels differing by at most k in each coordinate. The pixels differing by at most k form a square with side length $2k + 1$ since each coordinate differs by a value between $-k$ and k. (Formally, the pixels in this region are those whose L_∞ distance is $\leq k$. It's ok if those words don't make sense to you (yet).)