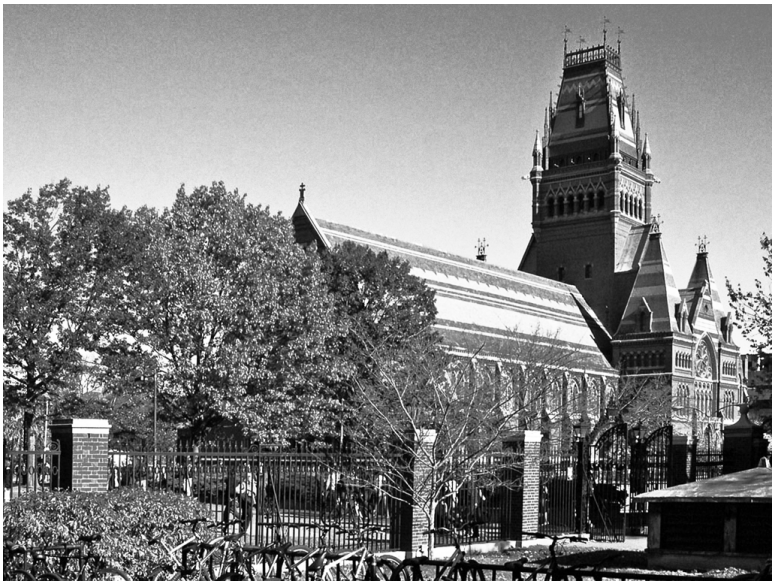
Problem 4 - CUDA Photoshop [35%]



In this problem, we will be manipulating images with a linear filter.

The code in `P4_serial.py` applies a sharpening kernel to the interior pixels of an image with a sharpening coefficient $\varepsilon = 0.005$ and the following 9-point stencil:

$$\begin{pmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{pmatrix}$$

The boundary pixels are left alone. This stencil acts to amplify edges in the image and some version of this is contained in nearly every photo editing software. The sharpening kernel is inherently unstable and too many applications will destroy the photo. One way to not go too far is to compute the mean and variance of the image and stop the application when the variance becomes too large.

Recall the mean is computed as:

$$mean = \mu = \frac{1}{width * height} \sum_{i,j} image[i,j]$$

And the variance is computed as:

$$variance = \sigma^2 = \frac{1}{width * height} \sum_{i,j} (image[i,j] - \mu)^2$$

In our case, we iterate until the variance has increased by 20%.

Implement With 2D Thread Blocks

Implement the sharpening kernel with PyCUDA using 2D thread blocks and a 2D grid of blocks. Every iteration, compute the mean and variance on the host before sending the image to the GPU for the sharpening kernel. Show that your program executes correctly. Find a good execution configuration (block and grid size) and explain why it is a good configuration.

Mean / Variance

Compute the mean and variance on the GPU as well. You may use any efficient method you like. Now you should be able to transfer the image to the device once before the loop and keep it on the device to compute the sharpening kernel and the mean / variance multiple times.

Your program should output the mean and variance every iteration like the serial version and save the final image as `Harvard_Sharpened_GPU.png` when complete.

HINT: Consider using a CUDA reduce and/or element-wise operations.

Benchmarking

Run your code with the following images:

- `Harvard_Small.png` (512x384)
- `Harvard_Medium.png` (1024x768)
- `Harvard_Large.png` (2048x1536)
- `Harvard_Huge.png` (4096x3072)

Analyze the time for setup (GPU memory allocation and initialization), the time to compute the mean / variance, and sharpening execution time. Compare the performance of the provided serial code with your GPU kernel.

Submission Requirements

- `P4.pdf`: Explanations.
- `P4.py`: Sharpening code with GPU kernel and variance computation.

Problem 5 - CUDA Photoshop 2 [30%]

Another interesting image processing algorithm is region growing. Suppose we want to find all of the "dark regions" of an image. Perhaps we want to extract shadows for shape matching, balance the dark regions with the light regions, etc. Our first instinct might be to simply filter the image for all the dark pixels. A quick script to do that might look like:

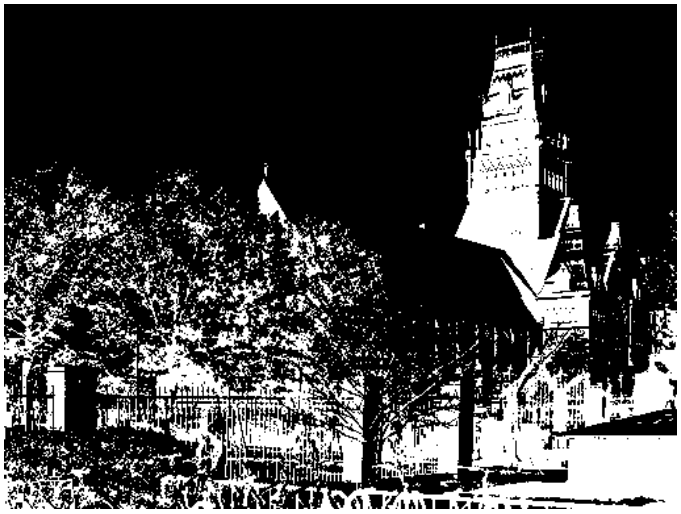
```
import matplotlib.image as img

in_file_name  = "Harvard_Small.png"
out_file_name = "Harvard_SimpleRegion_CPU.png"

# Pixel value threshold that we are looking for
threshold = [0, 0.27];

if __name__ == '__main__':
  # Read image. BW images have R=G=B so extract the R-value
  image = img.imread(in_file_name)[:,:,0]
  # Find all pixels within the threshold
  im_region = (threshold[0] <= image) & (image <= threshold[1])
  # Output
  img.imsave(out_file_name, im_region, cmap='gray')
```

and this results in



which is good... but... really noisy. We were really interested in the dark REGIONS and now we have a lot of individual pixels.

One solution to fix this is a seeded region growing algorithm. Instead of simply searching for pixels, we'll seed our search with a couple of "interesting" pixels and "grow" the region by repeatedly adding neighbors that satisfy our original threshold. This way, the chosen pixels must be connected to a seed pixel through some path of pixels that all satisfy our threshold. That is, these pixels are connected into regions like we desired.

We have provided `P5_serial.py` for seeded region growing on a grayscale image. First, all pixels with values inside a chosen *seed threshold* range are selected. Then, repeated grow operations add adjacent pixels with values inside the original *threshold* range to the image region. This process continues until the regions stop growing. The results are much more satisfying:

However, we may have made a mess of things. The original filter was trivial and embarrassingly parallel, but the region growing algorithm is much more complicated...

Full image region growing

Using the extreme parallelism of CUDA, we can let one thread take responsibility for one pixel on each iteration of the region growing algorithm. That is, the algorithm will look like

- Find seed points – embarrassingly parallel.
- Each thread takes one pixel of the image and determines whether it should be added to the regions.
- Continue until no pixels were added in the last iteration.

You must determine the criteria for step two and determine how to test for convergence (in a parallel-safe way) in step three. Your program should read in the largest of the images, find the seed points, apply the region growing operation, print the total run-time (including all communication), and save the resulting region image as `Harvard_RegionGrow_GPU_A.png`.

Queue-based region growing

The above approach works quite well and is a very "CUDA"-like solution as we use massive parallelism and most of the reads and writes can be coalesced. It is severely different from the serial version though. The serial version keeps track of only the pixels in the newly grown