Spring 2017: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 5 (due May 11, 2017)

Make your own homework choice. This final assignment gives you freedom to explore what you are most interested in. Please hand in (as always, using a git repository) a solution to either Problem 1 or Problem 2.

1. **Generalize the Multigrid Implementation.** Generalize the one-dimensional serial multigrid implementation[1] in at least one (you choose!) of the following directions:

   (a) Generalize to the two-dimensional problem. From previous homework, you already have implementation of the two-dimensional Jacobi method. Note that for Jacobi within multigrid, one should use a relaxation parameter ω in the Jacobi update step—see the one-dimensional version.

   (b) Extend either the one- or the two-dimensional version to a shared memory (OpenMP) parallel implementation and run a series of large problems on Stampede. Report scalability results.

   (c) Same but for distributed memory (MPI) version.

2. **Image convolution with OpenCL.** The convolution example code[2] implements an image blurring algorithm. This algorithms replaces each pixel's value with a weighted average of its neighbor pixels. Mathematically, a gray value image can be considered as a matrix with entries $p_{i,j}$ at the position (i,j), with $(0,0)$ being the upper left pixel and $i \leq W-1$ and $j \leq H-1$, where W and H denote the width and heigth (in number of pixels) of the image. The type of blurring is described by the blurring kernel[3] (which has nothing to do with an OpenCL kernel), which is a square matrix $K \in \mathbb{R}^{(2l+1)\times(2l+1)}$, where $l > 0$ is the half width of the blurring kernel. The blurred image pixels $\tilde{p}_{i,j}$ are then computed as follows:

$$\tilde{p}_{i,j} = \sum_{m,n=-l}^{l} K_{m,n} p_{i+m,j+n} \quad \text{for} \quad l \leq i \leq W-l-1,\, l \leq j \leq H-l-1.$$

   Here, we have used the indices $i,j \in \{-l,-l+1,\dots,l\}$ for the blurring kernel K, i.e., $K_{0,0}$ represents the center of the blurring kernel K. Moreover, blurring is only done l pixels away from the boundary, such that no boundary effects occur.[4] The code requires as input an image in uncompressed portable pixmap (PPM) format[5], as well as the number of repetitions of the blurring on the computing device, e.g.:

[1] https://github.com/NYU-HPC17/multigrid

[2] The example code can be obtained from https://github.com/NYU-HPC17/homework5. Note that this is a slightly simplified version of the problem we discussed in class, which is available in the lecture11 github repository.

[3] See for instance http://en.wikipedia.org/wiki/Kernel_(image_processing).

[4] The blurring kernel would otherwise have to be modified at the boundary.

[5] http://en.wikipedia.org/wiki/Netpbm_format

```
./convolution IMAGE.ppm 100
```

The output images are called `output_cpu.ppm` and `output_cl.pl`. There are several image viewers for the PPM format, for instance `gimp`, `Toyviewer` or even `emacs`. To convert an image from, say, JPEG format, into uncompressed PPM format on Linux, you can use:

```
convert -compress none IMAGE.jpg IMAGE.ppm
```

If you are using a Mac and the `convert` function is not installed, try finding an online tool that converts into PPM or use the example image that is checked into the repository.

(a) Pick your favorite image and convert it to PPM format. Run the convolution program on at least two different devices (besides your laptop/desktop, you can try `cuda1` or `cuda3` at CIMS, or you can use the GPUs on Stampede[6]) and report the number of processed pixels/s, the bandwidth, and the Flop/s. Also, show the original and the blurred image in your documentation. Try changing the local work group size (the size is currently set to 16×16 as defined at the beginning of the file `convolution.c`). Do you observe an improvement in the performance?

(b) As you can see from the code, the OpenCL application of the blurring operator is applied to the same input image many times, always resulting in the same output image. Change the program such that the blurring operator is consecutively applied to the input image, i.e., the kth application of the blurring operator is applied to the already $(k-1)$-times blurred image.[7] Try to avoid boundary effects[8], and document several output images, which result from different numbers of applications of the blurring operator to the original image. Note that after a large number of blurring applications, the image should become completely washed out, i.e., all pixels will (approximately) have the same gray value.

[6] https://portal.tacc.utexas.edu/user-guides/stampede#gpu-opencl

[7] One way to do this is to create a second OpenCL kernel, which copies the output image back onto the input image. This copy kernel is a simple modification of the blurring kernel. Surely, there are other possibilities to consecutively apply the blurring operator, for instance pointer flipping.

[8] Uncompressed PPM images are plain text file that contain that contain the matrix corresponding to the gray values of pixels. I found it useful to open the matrix in a text editor or import it into Matlab to compare the result after one, two or more applications of the blurring operator.