High-Performance Scientific Computing (MATH-GA 2011/ CSCI-GA 2945)

Homework Set 3

Out: September 26, 2012 · Due: October 3, 2012

This homework set lets you practice simple uses of OpenCL. There is an OpenCL implementation available on your virtual machine, but don't expect it to be blazingly fast. Since running on GPUs is a large part of the point of OpenCL, at the end of this assignment you can find some instructions for running your code on GPUs available on NYU's HPC clusters. For now, this is an optional (but highly recommended) part of this assignment. Since we'll be using these machines more over time, it'll be helpful to get started on using them now. In order to use them, you'll need an HPC account. To get one, you may need to request one at the account request page[1].

You may use the Lecture 4 demo code[2] as a starting point. (Make sure to look at the files ending in `-soln.c`, not the intentionally problematic 'initial' codes.

Problem 1: Image + OpenCL warm-up

Read in two numbers w and h from the command line and allocate three buffers `red`, `green`, `blue` on the compute device, each sufficient for $w \times h$ `unsigned char`s.

Write an OpenCL kernel that fills the buffers above with linearly blended colors such that

- The red channel is zero on the left-hand side of your image and 255 (the maximum channel value) on the right-hand side of your image.
- The blue channel is zero on the top of your image and 255 (the maximum channel value) on the bottom of your image.
- The green channel is zero everywhere.

Write your kernel with a work group size of 1×1 and a global size of $w \times h$.

Use the library routines provided in this repository[3] to write a PPM[4] ("portable pixmap") image file called `linear-blend.ppm`. The file `test-ppm.c` demonstrates the use of the library.

You may look at the resulting image by typing

```
display linear-blend.ppm
```

This command is preinstalled on your virtual machine. If you're not using the VM, find an image viewer that can display PPM images.

Note that images are conventionally represented in row-major order. In other words, the x coordinate varies fastest in memory.

Turn in a main C file `problem-1/blend.c` along with a kernel file `problem-1/linear-blend.cl`. Also turn in a Makefile that builds your code.

Also, please make sure to *not* check the generated image files into git. (You'll kill my server if you do. I'm not kidding.)

[1] `https://wikis.nyu.edu/display/NYUHPC/Request+or+Renew`
[2] `https://github.com/hpc12/lec4-demo`
[3] `https://github.com/hpc12/hw3-ppm`
[4] `https://en.wikipedia.org/wiki/Netpbm_format`

Problem 2: Compute a Mandelbrot set

The Mandelbrot set[5] is a subset of the complex numbers that has a surprisingly simple mathematical definition, but a surprisingly complicated mathematical structure. (Don't know complex numbers? Don't panic, see below.)

Somewhat formally, it is defined as

$$\{c \in \mathbb{C} : \text{The sequence defined by } z_0 = 0,\ z_n = z_{n-1}^2 + c \text{ becomes} > 2 \text{ in magnitude for some } n\}.$$

If your complex number skills are a bit rusty: A complex number consists of two real numbers (think `float`s) $z = (a, b)$. If we let $c = (x, y)$ and $z_n = (a_n, b_n)$ (for $n \geq 0$), then the above formula can be expressed as

$$a_n = x - b_{n-1}^2 + a_{n-1}^2$$
$$b_n = y + 2a_{n-1}b_{n-1}$$

a) Read in two numbers w and h from the command line and allocate three buffers `red`, `green`, `blue` on the compute device, each sufficient for $w \times h$ `unsigned char`s.

b) Write an OpenCL kernel that carries out the Mandelbrot iteration for $c = (x, y)$. Let x vary linearly from `xleft` and `xright` on the left and right, and let y vary linearly from `ytop` and `ybottom`, on the respective ends of your image. Pass these values as parameters to your kernel.

Use the following values for the work you turn in:

`xleft`	-2.13
`xright`	0.77
`ytop`	1.3
`ybottom`	-1.3

But feel free to play around.

For each pixel of the image, run at most `max_iter` iterations, where that is another parameter to your kernel. Stop the iteration when the square of the magnitude of the iterate reaches greater than four. For a complex number $z = (a, b)$, the square of the magnitude is computed as $|z|^2 = a^2 + b^2$.

Store the number of iterations until four is exceeded in all channels (red, green, blue) of your image, in such a way that if you used `max_iter` iterations, the value becomes 255. (You may use more 'interesting' color maps, too, if you like.) Store the resulting image as `mandelbrot.ppm`.

Write your kernel with a work group size of 1×1 and a global size of $w \times h$.

c) Modify your OpenCL kernel so that it uses a workgroup size of 16×16, but make sure that your program can still be run with any w and h. (I.e. you will have to handle edge cases.) Store the resulting image as `mandelbrot-large-wg.ppm`.

d) Use the `timing.h` infrastructure from the last two homeworks to time the execution of both the small-workgroup and the large-workgroup case. Make sure to wait for the compute device to finish its job in the right places. Output the performance in millions of pixels per second for each case, to three decimal places.

[5] `https://en.wikipedia.org/wiki/Mandelbrot_set`

e) Make sure you free/release all your buffers, command queues, host memory, and whatever other resources you've used.

   Also make sure that you check for error returns on all functions that can fail, including `malloc` and `OpenCL` interface functions.

Turn in a main C file `problem-2/mandelbrot.c` along with kernel files `problem-2/small-wg.cl` and `problem-2/large-wg.cl`, corresponding to parts b) and c). Make sure that `mandelbrot.c` exercises all parts of this assignment as described above when compiled and run. Also turn in a Makefile that builds your code.

Lastly, please make sure to *not* check the generated image files into git. (You'll kill my server if you do. I'm not kidding.)

High-Performance Scientific Computing (MATH-GA 2011/ CSCI-GA 2945)

Access to the NYU clusters

Logging into a cluster

All access to the high-performance machines at NYU goes through `hpc.nyu.edu` (a.k.a. the "bastion host"). You may log in using SSH[6]:

```
your-machine$ ssh NETID@hpc.nyu.edu
```

Your user name on this machine is your NYU NetID (the same one you use to log into NYUHome). Likewise, your password is the same one you use to access NYUHome.

Note 1 *As before, the dollar sign "$" represents a command prompt. The actual command you need to type follows after that. For clarity, a symbolic host name may precede the dollar sign.*

The virtual machine comes with `ssh` pre-installed. If you are using a Windows machine, you may use PuTTY[7].

The first time you log in, the system will warn you that it does not know about `hpc.nyu.edu`; just say "yes" to log in and permanently accept this machine as a known host.

From the bastion host, more or less the only thing you can do is use SSH to access other machines. To reach the "cuda" cluster, type

```
hpc$ ssh cuda
```

To reach the "Bowery" cluster, type

```
hpc$ ssh bowery
```

The first time you log in to each, you will again get a warning from `ssh`. Proceed as above.

The first time you log in, you will also be prompted to set up an SSH key pair. This key pair serves two purposes: First, it will allow your jobs to access the cluster's compute nodes on which they are supposed to run. Second, it will act as your key to the class collaboration space when logged in to the NYU clusters.

Just hit Enter when prompted for a passphrase. The interaction will look something like this:

```
It doesn't appear that you have set up your ssh key.
This process will make the files:
    /home/NETID/.ssh/id_rsa.pub
    /home/NETID/.ssh/id_rsa

Generating public/private rsa key pair.
Enter file in which to save the key (/home/NETID/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
```

[6] http://en.wikipedia.org/wiki/Secure_Shell

[7] http://www.chiark.greenend.org.uk/~sgtatham/putty/

```
Enter same passphrase again:
Your identification has been saved in /home/NETID/.ssh/id_rsa.
Your public key has been saved in /home/NETID/.ssh/id_rsa.pub.
The key fingerprint is: ...
```

Now you are logged into the cluster! More precisely, you will be logged into the cluster's so-called 'head node'. You may compile your programs there, but you are not supposed to run large computations on that machine. See the instructions below on how to start computation jobs.

Editing files on the clusters

A simple, friendly, somewhat bare-bones editor is available by typing

```
$ nano filename.c
```

Moving files to and from NYU HPC

See this wiki page[8] on how to move data back and forth between your computer and the HPC clusters. (Note that if you're moving source code, git via forge is usually more convenient.)

Note that if you have access to a machine that has SCP/SFTP accessible from the outside network (such as `access.cims.nyu.edu`), you can avoid the tedious two-step process via the bastion host by initiating file transfers *from* the clusters.

Using `git` on the clusters

To use `git` on 'cuda', you need to type

```
$ module load git/gnu/1.7.2.3
```

and on 'bowery', you need to type

```
$ module load git/intel/1.7.6.3
```

To avoid having to type this every time, consider adding this command to the `.bashrc` (note the dot!) file in your home directory.

NYU uses the software "Environment Modules[9]" to manage a variety of software installed on its clusters. By typing the '`module load`' command above, you have already used this system to allow you to use `git`. You may type

```
module avail
```

to find which other software modules are available. In particular, you may choose to use the Intel Compilers[10] instead of the default GNU ones. Like `git`, these compilers are activated through the `module load` commands. The NYU HPC Wiki describes these compilers in more detail.

Next, inform `git` of your name and email address, as before:

[8] `https://wikis.nyu.edu/display/NYUHPC/SCP+through+SSH+Tunneling`
[9] `http://modules.sourceforge.net/`
[10] `http://en.wikipedia.org/wiki/Intel_C%2B%2B_Compiler`

```
$ git config --global user.name "Your Name Here"
$ git config --global user.email you@yourdomain.example.com
```

Next, while you are logged into each clster, type

```
cluster$ cat $HOME/.ssh/id_rsa.pub
```

This will print a few lines (technically, just one wrapped line) that looks like this:

```
ssh-rsa AAAABw...vM3XdIbZWwmXH/iNbFWEhZw== NETID@login-0-1.local
```

Once you are logged into `forge`, click your name in the top left corner. Click "Update your account" on the left and then paste the line returned above into the field that says "Add public key". Click "Update your Account". The key update may take up to two minutes to fully complete, so if any steps involving git below fail, wait for two minutes, and then try again. This completes the setup steps that are only required once for each cluster.

Next, you may 'clone' an existing homework repository by using

```
git clone ssh://git@forge.tiker.net:2234/hpc12-hw3-netid123.git
```

You may make some changes here, do `git add` and `git commit`. `git push` will upload them 'to the cloud'. Likewise, `git pull` will download. That should help you get started. Git will also be next week's 'tool of the week' in the lecture.

Running code on the clusters

Here is a sample transcript that starts just after I logged in to the 'cuda' cluster:

```
# First, we're getting a list of all 'queues' supplied by the scheduler
# on 'cuda'.
[ak177@cuda ~]$ qstat -Q
Queue              Max   Tot   Ena   Str   Que   Run   Hld   Wat   Trn   Ext T
---------------- ---   ---   ---   ---   ---   ---   ---   ---   ---   --- -
serial             0     0   yes   yes     0     0     0     0     0     0 E
interactive        0     4   yes   yes     0     0     0     0     0     0 E
default            0     0   yes   yes     0     0     0     0     0     0 E
batch              0     0   yes   yes     0     0     0     0     0     0 E
p12                0     0   yes   yes     0     0     0     0     0     0 E

# Next, we're submitting an interactive job (-I) to a queue
# named 'interactive' (-q).
[ak177@cuda ~]$ qsub -q interactive -I
qsub: waiting for job 2113.cuda.es.its.nyu.edu to start
qsub: job 2113.cuda.es.its.nyu.edu ready

# Even if you loaded the git module on the head node, you may have to do this
# again now, because you're now logged into a new machine, a 'compute node'.
# Observe how the prompt has changed.
[ak177@compute-0-3 ~]$ module load git/gnu/1.7.2.3
```

```
# We'll grab the lecture demos...
[ak177@compute-0-3 ~]$ git clone git://github.com/hpc12/lec4-demo
Cloning into lec4-demo...
remote: Counting objects: 65, done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 65 (delta 34), reused 56 (delta 25)
Receiving objects: 100% (65/65), 30.94 KiB, done.
Resolving deltas: 100% (34/34), done.
[ak177@compute-0-3 ~]$ cd lec4-demo/

# ...and build them.
[ak177@compute-0-3 lec4-demo]$ make
gcc -c -std=gnu99 cl-helper.c
gcc -std=gnu99 -lrt -lOpenCL -ovec-add vec-add.c cl-helper.o
gcc -std=gnu99 -lrt -lOpenCL -ovec-add-soln vec-add-soln.c cl-helper.o
gcc -std=gnu99 -lrt -lOpenCL -oprint-devices print-devices.c cl-helper.o
gcc -std=gnu99 -lrt -lOpenCL -otranspose transpose.c cl-helper.o
gcc -std=gnu99 -lrt -lOpenCL -otranspose-soln transpose-soln.c cl-helper.o

# Let's see what devices we have here:
[ak177@compute-0-3 lec4-demo]$ ./print-devices
platform 0: vendor 'NVIDIA Corporation'
  device 0: 'GeForce GTX 285'
platform 1: vendor 'Advanced Micro Devices, Inc.'
  device 0: 'Intel(R) Xeon(R) CPU       E5405  @ 2.00GHz'

# Let's run the vector addition demo.
[ak177@compute-0-3 lec4-demo]$ ./vec-add-soln 10000000 10
create_context_on: specified device not found.
Aborted

# Ok, the code is still looking for the "Intel" platform, which doesn't
# appear to exist here. Let's change "Intel" to "NVIDIA":
[ak177@compute-0-3 lec4-demo]$ nano vec-add-soln.c

# Rebuild the code:
[ak177@compute-0-3 lec4-demo]$ make
gcc -std=gnu99 -lrt -lOpenCL -ovec-add-soln vec-add-soln.c cl-helper.o

# And off we go:
[ak177@compute-0-3 lec4-demo]$ ./vec-add-soln 10000000 10
0.001777 s
135.051488 GB/s
GOOD

# This command returns you to the head node and frees up the machine
# for other people. Since the machines are a shared resource, it's
# common courtesy to log out if you're not actively using the compute node
```

```
# you're logged # into. (It's fine to stay logged into the head node.)
[ak177@compute-0-3 lec4-demo]$ exit
```