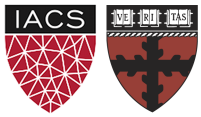
CS205, Fall 2015
Computing Foundations for Computational Science
Ray Jones



HARVARD
School of Engineering
and Applied Sciences

Homework 3
Due Nov. 18, 2015

This homework will be an exploration of OpenCL and GPU programming.

Get the HW3 skeleton files from GitHub **2**
Problem 1 - Get a functional OpenCL environment **3**
Problem 2 - Warm up: Mandelbrot set **4**
Problem 3 - Autotuning Kernels [25%] **5**
Problem 4 - 2D median filter [25%] **6**
Problem 5 - 2D Region Labeling [50%] **7**
Part 1: implement updates from neighbors 7
Part 2: fetch grandparents 7
Part 3: merge parent regions 8
Part 4: efficient grandparents 8
Part 5: no atomic operations 8

Get the HW3 skeleton files from GitHub

To bring your repository up-to-date with the CS205 homework repository, we will add it as a remote, then use it as the basis for your work on HW3.

First, switch to your local repository directory on your machine (or VirtualBox). Most likely, you can do this with this command:

```
cd ~/cs205-homework
```

Then, add the CS205 class HW repository as a remote with the name "upstream":

```
git remote add upstream https://github.com/harvard-cs205/cs205-homework.git
```

Fetch the changes we've made to that repository:

```
git fetch upstream
```

Check out a HW3 branch based on the upstream/master branch:

```
git checkout --no-track -b HW3 upstream/master
```

If you have uncommitted changes, you may have to deal with them. If you just want to throw them away, you can run "`git checkout -- .`", though you may just want to commit them all in a wrapup commit.

Make sure you are on the HW3 branch:

```
git branch
```

(You should see a list of branches with an asterisk next to `HW3`.)

Then, begin hacking, and `git add ...files...` and `git commit -m ''...message...''` as you go along.

When you are ready to submit your work, push it to GitHub into your remote repository (you can do this with partial results, if you want a remote backup of your work):

```
git push origin HW3
```

And then follow the instructions from Homework 0 to open a pull request from your HW3 branch to the CS205 homework repository.

Problem 1 - Get a functional OpenCL environment

If you have not already done so, clone a copy of the https://github.com/harvard-cs205/OpenCL-test-setup repository, and make sure `python test.py` produces useful output (i.e., does not crash). This repository is cloned from the Odyssey introduction materials; if you don't have a working installation of OpenCL on your own machine, you can log into Odyssey, clone the repository, and use `sbatch run_test.sh`.

Post on Piazza or contact `staff@cs205.org` if you are unable to get a working PyOpenCL setup.

Problem 2 - Warm up: Mandelbrot set

In `P2/mandelbrot.py` and `P2/mandelbrot.cl`, we've provided the skeleton for computing and plotting the Mandelbrot set, and computing the complex multiply-adds per second on whatever hardware you have access to. As a warmup, we suggest you add the missing code in `P2/mandelbrot.cl` and compare the compute power from OpenCL to that from AVX in the previous homework.

Submission Requirements

- None.

Problem 3 - Autotuning Kernels [25%]

In this problem, you will write two kernels to sum a vector of values, using slightly different strategies, and investigate tuning these kernels for different numbers of workgroups, and workgroup sizes.

As this is a bandwidth-limited problem, we will use a different approach than the one-thread-per-input-value, in that the total number of threads we launch will be far fewer than the number of elements we're going sum. Instead, each workgroup will be responsible for summing some (possibly noncontiguous) set of values into a partial sum vector that will be returned to the python code.

In `P3/sum.cl`, we've provided the skeletons for two kernels, with code in `P3/tune.py` to run these kernels, check their correctness, and measure their performance for different numbers of workgroups and group sizes.

Your first task is to add the missing code into `P3/sum.cl`.

In `sum_coalesced()`, when `get_global_size(0) == k`, the thread with `get_global_id(0) == i` should be responsible for adding the elements at $\{i, i+k, i+2k, ...\}$ up to the end of the input.

In `sum_blocked()`, when `get_global_size(0) == k`, the thread with `get_global_id(0) == i` should be responsible for adding the elements at $\{L \cdot i, L \cdot i + 1, L \cdot i + 2, ..., L \cdot (i+1) - 1\}$, where $L = \lceil N/k \rceil$ and N is the length of the vector. Be careful not to read past the end of the input vector.

Both functions should use a binary reduction in local memory to get the sum from within the workgroup. You can read about binary reductions here. This code can be identical in the two kernels.

After you complete the kernels, run the timing code and report in `P3/P3.txt` the best configuration for the machine you're using.

Submission Requirements

- `P3/sum.cl`: New version of the OpenCL kernels with the missing code added.
- `P3/P3.txt`: The best configuration and time for your hardware.
- Any graphs referenced in `P3/P3.txt`, if needed.

Problem 4 - 2D median filter [25%]

In this problem, you'll implement a 3x3 median image filter in OpenCL. We've given you the outline for how to solve this problem (in `P3/median_filter.cl`), and the relevant pieces (some here, some elsewhere), but have purposefully left most of the details unspecified and the skeleton empty.

First, download this file and save it as image.npz in the P4 directory.

For pixels at the image boundary, you should use the closest pixel in the valid region of the image.

Submission Requirements

- `P4/median_filter.cl`: Your finished median filter code.

Problem 5 - 2D Region Labeling [50%]

In this problem, you'll implement and optimize code for 2D region labeling. This is a common image processing operation, in which different foreground objects in an image are given a unique per-pixel integer label.

We have a provided a skeleton for this problem, in `P5/label_regions.py` and `P5/label_regions.cl`. You will only have to modify the second of these (though if running on Odyssey, you may want to remove the matplotlib-related code from the python file, or you can follow the *Setup X11 forwarding for lightweight graphical applications* instructions here and add "-x11=first" to the `sbatch` parameters).

The code we have provided initializes the label image, where each foreground pixel starts with its offset index within the image, and background pixels are given a value larger than any foreground pixel.

We've also provided code, in `propagate_labels()`, to load the label data into local memory, for each workgroup, with a 1-pixel halo, as well as write back the result of computations from `new_label` for the non-halo portion of this memory (i.e., the "core"), after any updates are made.

After each step below, run the code, record the number of iterations, and the average kernel execution time in your writeup.

Part 1: implement updates from neighbors

The first step is to make this program functional. Add code to compute, for each foreground pixel, the minimum of its 4 neighboring pixels and itself. This should only reference local memory, and should store the result in `new_label`. Each thread in the work-group should only compute `new_label` for its corresponding `(x, y)` in image coordinates, and should only read from local memory.

Note that values in local memory should be indexed relative to `buf_x` and `buf_y`.

When this algorithm runs to completion, each pixel should be labeled with the smallest label value that it can reach moving only through foreground pixels.

We have provided two images for testing, stored as numpy arrays, in `P5/maze1.npy` and `P5/maze2.npy`.

Part 2: fetch grandparents

In "union-find" problems, similar to this one, a common optimization is to replace each label with the label of its label (its "grandparent"). Since we initialize every pixel label with its linear index in the label array, we can perform this operation by replacing each value in `buffer` (at index `offset`) with `label[buffer[offset]]`.

Add this optimization after the fetch to local memory, but before neighbor updates, and write the value to `buffer` at the correct location. Operate only on core (i.e., non-halo) values.

Note that you will need to use `barrier(CLK_LOCAL_MEM_FENCE)` to ensure that the local buffer values have finished updating before the minimum neighbor calculation.

Part 3: merge parent regions

Another optimization is for child regions to merge their old and new parent whenever they change to a new label. Add this optimization, where each time a pixel changes from `old_label` to `new_label` (after grandparent fetching and minimum neighbor calculation), you update the global `labels[old_label] = new_label`.

Rather than making this assignment directly, use OpenCL's `atomic_min()` function to ensure no pixel's value in `labels` ever increases. You should also use `atomic_min()` when writing back the non-halo portion at the end of the computation. (An *atomic operation* both reads a memory location and optionally writes back a new value into it simultaneously, while preventing race conditions between threads.)

Part 4: efficient grandparents

Pixels within a workgroup are likely to have similar labels, since they are close together. When we fetch grandparents in part 2, we are likely to read the same location in labels across multiple work-group threads. On some GPUs, this will result in the reads being serialized, as well as repeated, which is inefficient.

To avoid some of the redundant global memory reads, modify your code for fetching grandparents to use a single thread in the workgroup. Have this thread remember the last fetch it performed, and avoid repeatedly reading the same global value repeatedly.

Explain, in terms of compute vs. memory, why using a single thread to perform this step is or is not a reasonable choice. Note that there is some variation in GPUs, so you may want to discuss your empirical results as well as speculate under what conditions those results might be different.

Part 5: no atomic operations

Atomic operations are also inefficient, and virtually guarantee serialization of memory access. Explain what would happen if instead of using the `atomic_min()` operation, one would use the `min()` function. Your explanation should consider whether the final result would still be correct, what might be the impact on the performance of the algorithm (time and iterations), could a value in `labels` ever increase, and could it increase between iterations?

Submission Requirements

- `P5/label_regions.cl`: Updated code after Part 4 (but including changes from previous parts) above.
- `P5/P5.txt`: Iteration counts and average kernel times after each change for Parts 1-4, an explanation for Part 4 as to why a single thread is a good (or bad) choice for this operation, and the explanation of Part 5.
- Any graphs referenced in `P5/P5.txt`, if needed.