Homework 9

Due: Apr 27, 2013 2:00 PM (early)
Apr 29, 2013 11:59 PM (on time)
Apr 31, 2013 11:59 PM (late)

This is a pair assignment, with all of the same rules as usual, except you *are* allowed to work with a partner you have previously worked with. You should already have a partner for this homework; if not, email the TA list soon.

Problems 1, 2, and 3 should be handed in electronically, by running `cs157_handin hw9-p1` or `cs157_handin hw9-p2`, or `cs157_handin hw9-p3`, and the other two problems should be handed in on paper.

Problem 1

(5 points)

If you have any suggestions for how to improve the course for next year, turn them in here in a text file for credit. As with the last homework, your grade here will be proportional to how much we appreciate what you turned in. More points will be given for suggestions you can convince us would most improve the course, that are feasible to implement, and that are suggested by few other people.

Problem 2

In this problem, you will go through some easy manipulations of an artificially generated "friendship graph". The aim is to understand random walks on graphs, and how they relate to eigenvectors.

We will consider a simple model where there are three groups of 1000 people, and for each pair of people, they are friends with probability 0.3 if they are in the same group, but only 0.2 if they are in different groups. We can represent this by a friendship matrix `f`, which has 3000 rows and columns, and where `f(i,j)` is 1 if person `i` is friends with person `j`, and 0 otherwise. (We assume the matrix is symmetric, as in, `f(i,j)=f(j,i)`; also, `f(i,i)=0`.) We can generate such a matrix in Matlab with the following command:

```
f=triu(rand(3000)<.2+.1*blkdiag(ones(1000),ones(1000),ones(1000)),1); f=f+f';
```

(You can look at this matrix with the command `spy(f)`, which displays nonzero entries in blue, though you may need to maximize the window or zoom in to see anything but blue.)

In this problem, we know the people 1 through 1000 form a group, 1001 through 2000 form the second group, and 2001 through 3000 form the third group. Even though we know this, we will try to recover this information *without* using this knowledge, to demonstrate some general techniques for analyzing graphs. In real life the groups will be mixed up.

1. (3 points) Fill in the template `randomFriendWalk` that generates a random matrix `f` as above, and then produces a matrix `fr` by dividing the entries of each column of `f` by the sum of the

entries in that column. Namely, `fr` will be a 3000×3000 matrix describing the transition probabilities for a *random walk on the friendship graph*. Specifically, if person i has k friends, then for each friend j of i, `fr(j,i)=1/k`.

If you start a random friendship walk at person `i`, and take one step, then your probability of being at each person is dictated by the 3000 element vector that is the ith column of `fr`; this equals the product of `fr` with the 3000 element column vector whose ith entry is 1. A random walk of length 2 can be generated by multiplying by `fr` a second time, etc.

More generally, if `p` is a probability distribution on people, namely a 3000 element column vector of numbers greater than or equal to 0, which sum to 1, then `fr*p` will be the probability distribution of starting from a person drawn from the distribution `p` and taking one random step along the friendship graph; `fr*fr*fr*p` will be the result of taking three random steps along the friendship graph, etc.

2. (5 points) We want to figure out who is in the same group that person 1 is in. (We *know* the answer is "people 1 through 1000", but we want to reconstruct this using only the friendship graph.) Consider random walks of lengths 1 through 10 starting at person 1. Construct the corresponding probability distributions `p1` through `p10` (which should be 3000-element column vectors of nonnegative entries, each summing to 1). Can you figure out from these probability distributions which people are in person 1's group?

   For this problem, you must fill out the template `findGroupByRandomWalk` so that it produces a single plot (using the `plot`) command, that clearly differentiates people 1 through 1000 from the rest of the people 1001 through 3000.

   (**Hint:** a single one of the distributions `p1` through `p10` probably will not tell you what you need to know; you should look at the difference between two of them. Which two are the best to compare? More specifically, after computing `p1` through `p10`, the entirety of your code for this part should be something like `plot(p1-p2)`, where "1" and "2" are replaced by better numbers.)

3. (2 points) The solution to the previous part might have seemed a bit arbitrary, and in particular the numbers in the y-axis of your plot might be as small as 0.000000001, indicating that this method will run into numerical precision problems for more complicated examples. Luckily there is something much more robust: plotting the 2nd and 3rd eigenvectors against each other. (The 1st eigenvector will be the distribution that the random walk converges to, times a constant; you can verify this by comparing with the vector `p10` you computed above.)

   Compute the first three eigenvectors of the transition matrix `fr` via `[v,~]=eigs(fr,3);` and plot the 2nd and 3rd eigenvectors against each other with `plot(v(:,2),v(:,3),'+')`. (The "`~`" in the first command means the second output of the `eigs` command is discarded; however, we cannot just use the one-output version of `eigs` because it has a different meaning.)

   Save your plot as an image and turn it in.

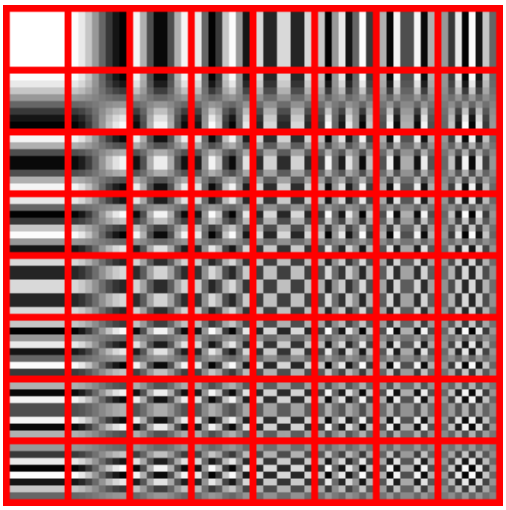
4. (5 points) Since this problem is so algorithmically straightforward, in order to benefit from it you need to think carefully about what it means and how you might use it in other contexts.

   Discuss why we put this problem on this problem set and what the take-home message is. Feel free to ask a TA about this. We *will* try to explain. (Or you can make use of Wikipedia and other resources for insight.)

Problem 3

In this problem we will try to find a better basis for images than the cosine transform basis used by JPEGs. We will fail, and that is interesting.

Recall that JPEG images express each 8×8 block of the image as a linear combination of 64 "2-dimensional cosine functions", where the $(0,0)$th cosine function, $\cos(0 \cdot x)\cos(0 \cdot y)$ is the constant function 1, the $(1,0)$th cosine function, $\cos(1 \cdot x)\cos(0 \cdot y)$ equals $\cos(x)$, and, for example, the $(i,j)th$ cosine function, for $i, j \in \{0, ..., 7\}$ is $\cos(i \cdot x)\cos(j \cdot y)$. A table of what all of these functions look like is in Wikipedia's JPEG article, at `http://upload.wikimedia.org/wikipedia/commons/2/23/Dctjpeg.png`, and is included here.



In this problem, we will be trying to redesign this part of JPEG from scratch, using what is essentially "principal components analysis". So that we can more easily see the patterns that develop, we will use 16×16 image blocks instead of 8×8 blocks, and we will thus need to find 256 of them so that we can express all possible 16×16 image blocks.

We will tailor our blocks to a particular image. Pick a high-quality JPEG image from a natural source, and load a matrix of grayscale values into Matlab. For example, you could make use of Bilbo with `im=mean(double(imread('hobbit.jpg')),3);`

1. (10 points) Fill in the template `betterImageBasis` and submit an image along with it so that your code loads the image (without crashing when we try it), and then proceeds as described below.

   Write code that constructs a 256×10000 matrix `M`, each of whose columns consists of the 256 elements of a *randomly chosen* 16×16 block of the image. Your 16×16 blocks should *not* be aligned to only start at multiples of 16 pixels but should be chosen so that starting at each location $x = 1, 2, 3, 4, 5, \ldots$, $y = 1, 2, 3, 4, 5, \ldots$ is equally likely. Note: to reshape a 16×16 submatrix s in Matlab to be 256×1 you can use `r=reshape(s,256,1);`.

   Then compute the singular value decomposition of this matrix `M` via `[u,~,~]=svd(M,'econ');`

   (Note that we could have instead computed `u` as the eigenvectors of $M \cdot M^T$, with `[v,~]=eig(M*M');` which will produce the same result as `u` but in different order, though the `svd` method is more numerically robust.)

The matrix `u` should now be 256×256, where each column represents a 16×16 image chunk (which you can reshape to be 16×16 with another call to `reshape(...)`); where collectively all 256 image chunks (columns) can be added together in different combinations to represent all 16×16 image blocks; and where the first column is the most important image chunk, the second column is the 2nd-most-important, etc. Namely, if you had to pick only 10 coefficients $c_1, \ldots c_{10}$ to represent an image, you should represent an image as c_1 times the first chunk, plus c_2 times the second chunk, etc. This is the significance of the "principal components analysis" we have done.

Generate a 256×256 image, consisting of these 256 16×16 blocks, arranged in 16 rows of 16 columns, starting from the top left and going to the right in descending order of importance.

2. (5 points) Since this problem is so algorithmically straightforward, in order to benefit from it you need to think carefully about what it means and how you might use it in other contexts.

   Discuss why we put this problem on this problem set and what the take-home message is. Discuss how the image you generated compares to the cosine basis image. Feel free to ask a TA about this. We *will* try to explain. (Or make use of Wikipedia and other resources.)

Problem 4

In this problem we will see why max-flow is such a powerful algorithmic technique.

1. (15 points) Consider a version of the "maximum matching" problem, where there are $n \geq 100$ people on the left, 100 people on the right, certain pairs of them are willing to get married ("matched"), no one can get married twice, and we want to match the maximal number of people so that we end up with 100 marriages.

   Recall from class that you can set this up as a max-flow problem, by considering the people as a graph with 1 unit of capacity along each edge connecting a person on the left to a person they are willing to marry on the right; additionally, each person on the left has an edge from the source s with capacity 1 (representing the constraint that each person on the left can have at most 1 marriage), and each person on the right has an edge going to the sink t with capacity 1 (representing the constraint that each person on the right can have at most 1 marriage). The algorithm works by first running one of the maximum flow algorithms we have seen in class, and then marrying any two people that have flow between them.

   For this part, modify this construction so that it solves the problem with the additional constraint: the first 40 people on the left *must* get married. Prove that your construction is a correct algorithm.

   Please include a diagram. Your proof should be sure to touch on the following points: 1) any assignment returned by your algorithm is a valid marriage plan; in particular, be sure to point out that integer capacities on edges implies that the flow returned by the algorithms from class will have integer values; 2) any marriage plan could be converted into a flow that satisfies the constraints you describe, proving that your algorithm does not "miss out" on any good marriage plans.

2. (10 points)

   Consider the (apparently very different though actually very similar) problem of scheduling weekly TA office hours in a course. Suppose there are n TAs on the course staff and m

non-overlapping one-hour slots throughout the week during which TAs may hold office hours; however, no more than one TA can be assigned to each hour. Each TA i is available only during some subset $T_i \subseteq \{1, \ldots, m\}$ of the weekly slots; further, each TA has to hold at least 2 hours every week. Your task is to assign TA staff to TA hours in a way that satisfies all aforementioned constraints.

Hint: Design your algorithm as an adaptation of your algorithm for the previous part, using a max-flow algorithm as a subroutine (how do each of the elements of this problem correspond with elements from the problem of the previous part?). You might want to first consider the problem without the constraint that each TA must cover at least two hours.

Include a proof of correctness, though feel free to make use of concepts from the previous part to avoid repeating yourself.

Also, analyze the run time of your algorithm here in terms of n and m. You might want to check `http://en.wikipedia.org/wiki/Maximum_flow_problem#Solutions` for reference on the running time of max-flow algorithms.

Problem 5

In this problem you will be designing an algorithm to transpose a 4096×4096 ($2^{12} \times 2^{12}$) matrix of 4-byte entries on a graphics card. You will write pseudocode for this, and analyze it in enough detail to confidently predict its performance when run on thousands of threads in parallel on a detailed model of a modern Nvidia graphics card. See the slides from `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf` as covered in lecture for examples.

The reason you cannot simply copy each entry $A[i + 4096 \cdot j]$ of the matrix to its location in the transpose $B[4096 \cdot i + j]$ is that if the input matrix is accessed in row-major order, the output matrix will be accessed in column-major order, and one of these two orders will be very wasteful of memory resources.

This homework assumes knowledge from lecture. See the appendix at the end of this document if you've forgotten any terms used in the descriptions below.

Important Time Constraints

- *Global memory*: It takes 200 ns to access one 128-byte page of global memory.
- *Shared memory*: It takes 1 ns to access 4 bytes (one element) from one bank of shared memory.
- *Registers*: Registers are free.

Important Size Constraints

- There are 8 multiprocessors, each of which can process one thread block at a time. A block can have up to 32 warps. A warp contains 32 threads.
- Each multiprocssor has 32 KB of shared memory, which is broken up into 32 interleaved banks of equal size.

- There is room for 64K (2^{16}) 4-byte registers per multiprocessor, though you should not need most of this.

Questions

1. (2 points) Suppose we used only 1 multiprocessor and only 1 thread. How long would it take to read in (and use) 128 bytes from global memory? How long would it take if we used 1 warp instead?

2. (2 points) If a block contains 32 warps, how much memory in registers can each thread use? How much shared memory can we dedicate to each thread?

3. (a) (2 points) How long would it take for 1 warp to access 32 4-byte elements from the same bank of shared memory?

   (b) (2 points) How long would it take to access 32 4-byte elements if each element is in a different bank?

4. (6 points) Write pseudocode to read in a 32×32 matrix from a $32 \times 32 \times 4 = 4096$ byte block of global memory `g_I` to shared memory, and then output its transpose to a block of memory `g_O`. Your code must: 1) use 1 block of $32 \times 32 = 1024$ threads, 1 thread per entry of the matrix; 2) each time a warp of 32 threads reads or writes global memory, the 32 threads must collectively touch exactly one 128 byte block of global memory. (You may assume that the blocks `g_I` and `g_O` are aligned to 128-byte boundaries.)

   You should write one function, which takes as input two locations in the graphics card's global memory: the input array `g_I` of 1024 single-precision floating point numbers (each taking 4 bytes of memory), and an output array `g_O` of the same size.

   Hint: You have to make use of shared memory; also, don't forget to surround any shared memory writes with `syncthreads()` calls, if you expect other threads to be reading from this location. See the appendix for more details on writing pseudocode for graphics cards.

5. (6 points) Write pseudocode as above but with the additional constraint that each time a warp of 32 threads reads or writes *shared* memory, the 32 threads must collectively touch each of the 32 banks of shared memory exactly once.

6. (6 points) Solve the original problem subject to the rules of the last two parts: modify the pseudocode of the last part to construct the transpose of a 4096×4096 matrix. You should write one function, which takes as input two locations in the graphics card's global memory: the input array `g_I` of 4096×4096 ($2^{12} \times 2^{12} = 2^{24}$) single-precision floating point numbers (each taking 4 bytes of memory), and an output array `g_O` of the same size.

   You must specify the *execution configuration*, which consists of two numbers describing how the code should be run in parallel: the *block size*, which specifies how many threads are in a thread block, and the *number of blocks*.

7. (8 points) Motivate, explain, and analyze your pseudocode for the last three parts to convincingly demonstrate that you have solved the problem.

8. (6 points) Estimate how long your final matrix transpose code will take to run, using the following parameters:

- Any request to global memory will take at least 200ns (one nanosecond, abbreviated "ns", is 10^{-9} seconds), though other warps on a multiprocessor may make progress while one warp is waiting for memory
- The total bandwidth to global memory is 200GB/s across all multiprocessors
- Each bank of shared memory can transfer one 4-byte element per nanosecond. (Recall that each multiprocessor has 32 banks of shared memory, and there are 8 multiprocessors.)
- Otherwise, assume that at each nanosecond, for each block of 32 (out of 128 total) cores in a multiprocessor, a warp of threads will start execution of a new instruction.

Note: Since there is a limit of 200GB/s on memory transfers, and our matrix takes $4096 \times 4096 \times 4$(bytes per element) $= 64MB$ of memory, and transposing this matrix involves both reading it, and writing its transpose, this cannot possibly happen in less than $\frac{128MB}{200GB/s} = 0.00064$ seconds. It would be nice if your code were close to this fast, though this question only asks you to accurately analyze the performance of the code you have written, whether or not it is fast. However, the fact that each warp of your pseudocode accesses both global and shared memory in the most efficient possibly way, combined with a high number of threads to help ensure that each multiprocessor always has something to do, means that your code should run very fast, or at least should be easy to tweak into code that runs very fast! (Recall that "MB" in some contexts means 10^6 bytes, in other contexts means $2^{20} = 1.048576 \cdot 10^6$ bytes; we are not worried about this distinction in this assignment.)

Appendix

Memory model

Each group of 32 threads (a *warp*, as defined below) has its memory accesses dealt with separately. Global memory ("RAM") can only be accessed in 128 byte chunks, which correspond to exactly 32 4-byte entries. Namely, if a warp of 32 threads simultaneously accesses memory locations 0 through 31 of a single-precision (4 bytes per element) array A, then a single 128 byte memory transfer will occur; if these threads instead access locations 20 through 51, then two 128 byte memory transfers will occur, one for locations 0 through 31, a second for locations 32 through 63, even though half of these memory transfers are wasted. If these threads instead access locations 0, 100, 200, 300, ... 3100, then $32 \times 128 = 4096$ bytes of memory must be transferred while the 32 threads only make use of 4 bytes of memory each, at an efficiency of $\frac{32 \times 4}{4096} = \frac{1}{32}$.

Crucial parameters: each global memory access has a *latency* of at least 200ns (0.000 000 2 seconds) between when it is requested and when it is delivered to the thread. In addition, the total rate at which memory may be transferred, to and from all threads on the graphics card, is at most 200GB/s (gigabytes per second).

In addition to global memory, there is also *shared* memory, and *registers* (see below for definitions). Registers you may consider free to access.

Shared memory consists of 32 interleaved *banks*, storing 4 byte entries. If you are storing a single-precision (4 bytes per element) array A in shared memory, then entries 0, 32, 64, 96, ... are stored in bank 0; entries 1, 33, 65, 97, ... are stored in bank 1; entries 2, 34, 66, 98, ... are stored in bank 2, etc. Crucially, each bank of shared memory may transfer one 4-byte entry per nanosecond. Thus if a warp of 32 threads accesses entries 0 through 31, this can be completed in one nanosecond. However, if a warp of 32 threads accesses entries 0, 16, 32, 48, 64, 80, 96, ..., then this will make use of only banks 0 and 16, and each of these two banks will have to transfer 16 entries each, which will thus take at least 16 nanoseconds to finish.

Hardware model

Our graphics card consists of 8 *multiprocessors*, each of which contains 128 *cores*, each of which execute code at 1GHz (billion instructions per second). Each thread block is executed on a single multiprocessor. Each multiprocessor contains 32KB (2^{15} bytes, or 2^{13} single-precision values) of shared memory, and 2^{16} 4-byte registers. Registers store values private to a single thread, including loop indices like i and j, or intermediate values of computations. The number of registers used by your code times the number of threads in a block must be less than 2^{16} in order for your code to be able to run.

Global memory is visible to all threads; shared memory is visible to only those threads in the same thread block; registers are only visible to a single thread. Namely, if your code stores variable i in a register, then each of the threads will have a separate version of i; if your code stores i in shared memory, then each thread *block* will have a separate version of i; if your code stores i in global memory, then all threads will share the same i.

The number of thread blocks can (and often will) be larger than 8 (the number of multiprocessors), in which case the thread blocks are executed in some order, one per multiprocessor, until all thread blocks have been processed.

A thread block is processed on a single multiprocessor, one instruction at a time, in groups of 32 threads know as a *warp*. The number of threads in a thread block may be as large as 1024 (2^{10}), and in particular may be larger than the number of cores on a multiprocessor (128). If the number of threads in a block is less than 128, then some cores in the multiprocessor will be idle. At each nanosecond, the multiprocessor chooses 4 warps of 32 threads, and starts the execution of the next instruction from each warp, in parallel on its 128 cores. Some instructions, such as global memory accesses, will take a long time to return their value; this architecture makes efficient use of that time by allowing more warps to be executed while waiting for the results from instructions on other warps. If the `syncthreads()` instruction appears, then all threads in the thread block will wait here until every thread has reached this point.

Coding for graphics cards

Your code should be standard pseudocode, to be run by each of the (*block size*) × (*number of blocks*) threads, with the following additional abilities: your code can ask for the block size, it can ask for the number of blocks, it can ask for its *thread number* (a number between 0 and (*block size*) − 1), and its *block number* (a number between 0 and (*number of blocks*) − 1). Every variable must be stored either in global memory, shared memory, or in a register, and you must specify this in your pseudocode (anything you do not specify will be assumed to be stored in a register); arrays taken as input or output to your function will be located in global memory. Finally, there is exactly one synchronization primitive your code may call: `syncthreads()` "waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `syncthreads()` are visible to all threads in the block."

Consider the sample psuedocode below:

```
someFunc(array inp, array out, width, height)

    register xblock = blockNumber()*blockSize() mod width
    register yblock = floor( blockNumber()*blockSize() / width )
    register tid = threadNumber()
    register sid = xblock + yblock*width +tid

    out[sid] = inp[sid]; //400ns per warp
```

What is this pseudocode doing? (Yes, it is trivial)