Homework 9: Countdown to the ICO

Due: Dec 4, 2018 6:00 PM (early)
Dec 7, 2018 6:00 PM (on time)
Dec 9, 2018 6:00 PM (late)

This is a partner homework as usual: work on all problems together; come to all office hours together; you are responsible for everything you and your partner submit and it is an academic code violation to submit something that is not yours. Check for your partner in the Google doc that was sent out. **If you don't have a partner, please let us know.**

Make sure that you clearly indicate on each problem which track you're in (either **1-credit**, or **full**). **1-credit track:** do problems 1, 3, 5 *and indicate "***1-credit***" on each problem.*

Problem 1

AR/VR applications galore! Color in this board! It's like Magic Leap!

(15 points)

Consider a subset S of squares on an $n \times n$ grid (you can think of S as being some shape on a chessboard, for example). Your goal is to color as many squares in the shape S as possible, subject to the constraint that in any row or column, at most k squares can be colored.

Find a polynomial time algorithm for this problem (polynomial in n and/or k).

Problem 2

Hey coder person! Help me design my 3d-printing algorithm now!

(15 points)

Consider the 3-dimensional version of the previous problem: the board consists of certain cubes from a 3-dimensional grid and a number k, and the challenge is to color as many cubes as possible subject to the condition that no more than k cubes in any "x-slice", "y-slice" or "z-slice" are marked—an "x-slice" is a 2-dimensional plane specified line specified an x coordinate, and consists of all cubes with coordinates $(x, *, *)$ for any values of "$*$"; "y-slice" and "z-slice" are defined similarly.)

Your challenge: show this problem is in fact NP-hard.

We strongly suggest trying to prove this from the following NP-complete problem: "3-way matching". Analogously to bipartite matching, there are now 3 sets that need to be matched up, and certain compatible triples. For example, a company has certain jobs that need to be done, certain employees to do them, and certain offices available for the employees to work in. Certain triples—given as input—are possible, for example: "Alice is willing to work job X if she gets office 101", "Bob is willing to work job Y if he gets office 99", etc. It is NP-complete, given such a list as input, to find the maximum number of triples you can match, where no person, job, or office can be matched more than once.

Problem 3

It's like Tinder, but for TA hours!

1. (10 points) Consider a version of the "bipartite maximum matching" problem, where there are $n \geq 100$ people on the left, exactly 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 end up with 100 marriages (the "maximum matching").

   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, 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.

   If you want to think about this problem without a hint, stop reading now. Otherwise: the point is that this graph flow terminology makes it is very easy to express upper bound constraints, just by adding capacities appropriately, however lower bound constraints, such as the constraint that the first 40 people *must* get married, seem hard to capture. The key insight is that lower bounds on those 40 people can be expressed as upper bounds somewhere else. 40+60=100. Namely, at most 60 marriages can occur among all the *other* people on the left.

2. (10 points)

   Consider the (apparently very different though actually very similar) problem of scheduling weekly TA office hours in a course. The course has decided it must hold exactly H office hours every week. 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.

Problem 4

It's like Tinder, but for finding a job at a start-up!

(20 points total) At a job fair, there are n students, and m recruiters, and certain pairs of students and recruiters want to meet for an interview. There are 10 time slots in which interviews can occur, numbered 1 through 10, and we want to arrange a schedule of interviews so that **every** desired interview is on the schedule. Each student or recruiter can only be in 1 interview at once. If some student or recruiter has more than 10 interview requests, then scheduling interviews with only 10 interview times is clearly impossible; however if all students and recruiters have at most 10 people they each want to meet, then your task is to prove a) an interview schedule is possible, and b) you can find it in polynomial time.

1. (5 points) Prove that you can schedule a *single* round of interviews that gives *every* student an interview, **if** the following condition holds: there is no set S of students such that the total set of recruiters they want to meet, across all their requested interviews, is smaller than S. (**Hint:** represent this as a max-flow problem, use the fact that "max-flow equals min-cut", and then show that any cut of capacity less than n can be transformed into a set of students S where the total number of recruiters they want to meet is smaller than S.)

2. (8 points) Consider the simplified case where $n = m$ and **every** student and recruiter has **exactly** 10 interview requests. Show that there exists a "perfect matching" between students and recruiters, namely that in a single round of simultaneous interviews, you can schedule everyone an interview with no conflicts.

3. (2 points) Given the nice structural result you proved above for the simplified case where every student and recruiter has exactly 10 interview requests, now show how to transform the original problem to this case. Namely, given an arbitrary list of interview requests such that no one is involved in more than 10 interviews, describe a polynomial time algorithm that creates imaginary interview requests, and, optionally, imaginary students and recruiters, such that the new total number of interview requests for every (real or imaginary) person equals exactly 10. (**Note/Hint:** it is easier if you allow student-interviewer pairs to request multiple interviews with each other.)

4. (5 points) Fill in the remaining details of your algorithm and prove its correctness: show how to schedule interviews in the 10 slots so that every interview request is satisfied. (Do not write a maximum matching algorithm here; just refer to the max-flow algorithm from class. Fill in the details explaining how to use this algorithm to solve the interview scheduling problem.)

Problem 5

My bitcoin mining app isn't fast enough! Help me make it faster—but you better not want equity!

(30 points total) 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 (remember homework 3 problem 2).

Finding the right way to do this on a graphics card requires understanding the different parameters of a graphics card's performance and how they fit together. The basics are below; see the appendix at the end of this document for a more complete introduction to the graphics card model of computing.

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 exactly 32 threads, which all run in lockstep (each of the 32 threads in a warp runs the same code, at the same time, though possibly on different data). Each multiprocessor has 128 cores, and thus at each nanosecond, a multiprocessor can make progress on up to 4 warps at the same time.
- Each multiprocessor has 32 KB of shared memory, which is broken up into 32 interleaved (striped) 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 128 bytes from global memory, assuming that each load request must be serviced before the next request can be made? How long would it take if we used 1 warp instead?

2. (2 points) If a block contains 32 warps, how many registers can each thread use? How much shared memory can we dedicate to each thread? (Shared memory is shared across all threads in a thread block, but it is often useful to think in terms of ratios like "memory per thread".)

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

   (b) (1 point) How long would it take for 1 warp to access 32 4-byte elements if each element is in a different bank?

4. (5 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()` synchronization barriers, if you expect other threads to be reading from this location. See the appendix for more details on writing pseudocode for graphics cards.

5. (5 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. (This should require a small but clever modification to your solution for the previous part, tweaking how you use shared memory, and possibly also how much you use.)

6. (4 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. (5 points) Motivate, explain, and analyze your pseudocode for the last three parts to convincingly demonstrate that you have solved the problem.

8. (5 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 known 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. Explicitly: if warp A executes a load from global memory into a register x, then execution of warp A cannot continue until register x (for each of the 32 threads in warp A) gets filled from memory, which will take at least 200ns; crucially, *the graphics card will continue work despite the stalled warp A, by finding some other warp B that can execute an instruction, and executing its next instruction.* All this happens every nanosecond. The converse situation, where warp A *stores* values to global memory does not stall the warp, because (even though memory may not be written for another 200ns) the processor does not need to wait for any new information to continue execution.

If the `syncthreads()` instruction appears, then all threads in the thread block will wait here at this "synchronization barrier" until every thread has reached this point and all memory accesses are complete. Synchronization is a vitally important part of parallel programming. CUDA makes it very easy to synchronize within a thread block (running on a single multiprocessor), but strongly discourages any attempt to synchronize across the 8 multiprocessors.

Coding for graphics cards

Your code should be standard pseudocode, to be run by each of the (*block size*) $\times$ (*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."

Recall the pseudocode for adding two vectors, one entry per thread.

```
vecAdd(A,B,out)
  register i = getMyThreadIndex() + getMyBlockIndex() * getBlockSize();
  out[i] = A[i] + B[i];
```