Chase Jones

Dr. Muyan

CSC 196u

8 October 2018

Assignment #1 Questions

Problem #1 Questions

1. The maximum number of threads that can be launched is 33,553,920.

2. The programmer may not want to launch the maximum number of threads if they were doing computations on an incomplete block.

3. Lack of data to operate on may limit the program from launching the maximum number of threads, in order to increase computation time spent on other tasks. It'd be silly to launch 2048 threads for something that only required 4. That's 2044 threads that could have been working on something else.

4. Shared memory is memory on the SM (streaming multiprocessor) that can be accessed much faster than global memory. It also allows threads to communicate with one another.

5. Global memory is the general memory of the graphics card, and is much slower than shared memory. Global memory can also cause data hazards if the threads are not synced correctly.

6. Constant memory is memory that will not change over the life of the kernel. This is for things like the grid and block sizes and other constants to be used during kernel launch.

7. Warp size is the number of threads an SM can execute concurrently.

Problem #2 Questions

1. The number of floating point operations being performed by the vector add kernel is simply N additions, since the kernel only really has 2 lines. The first line is an integer addition to get an offset for the float *. The second line is a bit more complex. It does two integer additions to get offsets for the

float elements of the float *, then it adds the two floating point operations. The kernel then exits. Therefore, it only executes N floating point operations (C = A+B).

2. The kernel reads 2N times from the global memory, as it has to read every value from one vector then every value from the other vector.

3. The kernel writes N times to the global memory, one time for each value computed. In a vector of size N, it must write N elements.

4. Changing colorspace of a photograph by a linear value for RGB, probabilities in statistics, book-keeping.

Problem #3 Questions

1. For every value in the output matrix, it is a result of (numAColumns^2)*2 floating point operations (multiply value in A by value in B, then add result to Pvalue). Therefore the kernel must perform numCRows*numCColumns*2(numAColumns^2) floating point operations to receive the product.

2. For every value in the output matrix, it is a result of numAColumns^2 global memory reads (numAColums per row/column per input Matrix). Therefore, the kernel must perform numCRows*numCColumns*(numAColumns^2) global memory reads to receive the product.

3. For every value in the output matrix, the kernel must perform numCRows*numCColumns global memory writes.

4. The only way I can think of to optimize the algorithm would possibly being able to store the global memory reads for later use by different threads, so they don't have to be loaded many times over; such as is done in the TiledMatrixMulitiply problem.

5. Transformations in graphical programs, linear algebra, physics calculations.

Problem #4 Questions

1. For every value in the output matrix, it is a result of (numAColumns^2)*2 floating point operations (multiply value in A by value in B, then add result to Pvalue). Therefore the kernel must perform numCRows*numCColumns*2(numAColumns^2) floating point operations to receive the product.
2. For every value in the output matrix, it is a result of (numARows*numAColumns+numAColumns*numBColumns)*((ceil(numAColums/TILE_WIDTH)+1)^2) global reads, since all elements are read into Shared memory once per tile, then used by the other threads accessing the shared memory.
3. For every value in the output matrix, the kernel must perform numCRows*numCColumns global memory writes.
4. Further optimization might require making the shared memory / tiles larger to accomodate more values, and less global memory reads.
5. A lot of errors can be introduced from the tiling aspect: for instance, boundary checking is a must to make sure values aren't being incorrectly set. Another difficulty is using many different variables to reference specific locations, such as all the ty, tx, TILE_WIDTH, Row, Col, and the various boundary sizes.
6. In this case, you'd probably want to split up the memory writes, so some element may only calculate half a row * half a column, and another grid/block take up the computation. Although in this case, you'd want to add to the value in the output matrix, rather than replacing it. Therefore, it's probably a good idea to set all output matrix values to zero first.
7. In this case, it'd be much like the first. You'd have to break the matrices up into smaller pieces which are all initialized to zero, do some calculations, add the values back, and continue with the next grid. It'd be very intensive on the system, but if something can't fit into memory, the only option is to break it up.