Julian Singkham

1. How many floating operations are being performed in the matrix multiply kernel that uses shared memory? Explain.

There are 2*m*k*n floating point operations. There are a total of m*n*k multiplication operations as the output matrix is of size m*k and there are n number of multiplications per output element. There are a total of m*k*n number of addition operations as the output matrix is of size m*k and there is n number of addition operations per output.

2. How many global memory reads are being performed by the kernel that uses shared memory? Explain.

There are (m*k)*2*ceil(N/T) number of global memory reads being performed. M*K is the number of elements of the output matrix and 2*ceil(N/T) is the number of global memory reads per element as each element needs to read from matrixA and matrixB.

3. How many global memory writes are being performed by the kernel that uses shared memory? Explain.

There are (m*k) number of global memory writes. The only time the shared memory kernel writes to global memory is when it saves the an element of the output variable.

4. Describe what further optimizations can be implemented to the kernel that uses shared memory to achieve a performance speedup.

You could base the kernel on computing the row of the output matrix. Each block would compute the output row, with the row of matrix A and matrix B being in shared memory. The first row in matC all use the first row of matrix A so this reduces the amount of global memory calls as matA can stay in shared memory. Each element corresponds to a different column of matB so by saving the row of matB you can calculate numerous output elements at once.

5. Compare the implementation difficulty of the kernel that uses shared memory compared to the kernel without shared memory. What difficulties did you have with this implementation?

The main difficulty with the shared memory implementation is checking for boundary conditions, specifically the if statement conditions. Placing the boundary conditions itself wasn't difficult as you add them to every memory call. Knowing what to put inside the condition statements took a bit of trial and error as element traversal is a difficult topic for me.

6. Suppose you have matrices with dimensions bigger than the max thread dimensions. Sketch an algorithm that would perform the multiplication in this case.

- The scheduler would compute the number of blocks its able to, and once a block is finished it creates a new block that will cover the next output element.
- Let's say the the GPU can only handle 1 block of 16x16 threads.
- The block would compute 256 elements of matA (row) * 256 elements of matB (col).
- The block would loop back and cover the next 256 elements until they have reached the last element in the row in matA.
- The next block then repeats the process on the next col of matB.
- Repeat the above process on the next row of matA

7. Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would
Create variables that record the last accessed element of matA, matB, matC
Loop until all elements of matA and matB have been used

- Partially load matA, matB, and matC upto the global memory limit
    - Load the max number of columns in matB, then load the max number of rows in matA, then load the max number elements in matC.
        - Starting element will be based on the last recorded location of matA, matB, matC
        - Record the location of the last element used in matA, matB, and matC
    - Worse case scenario
        - MatA loads number of elements in row = MatB number of elements in col
        - MatC loads number of elements = number of complete row*col groups
            - Worst case being only 1 element
- Perform the multiplication
- Save the partial output matrix to host

The kernel would not need to be modified as the host would take care of loading the correct matrices to the GPU.

Julian Singkham

GPU & GPU Tiled Implementation Test

Testing with input of 32768, 16384, 8192

```
singkhamj@dh-node2:~/cuda/build/lab4$ ./TiledMatrixMultiplication 32768 16384 8192
**************Initializing CPU Variables**************
**************Initializing GPU Variables**************
*****************GPU Implementation*****************
Calculated Output = 16384.000000
True Output = 16384
Output Difference = 0.000000

GPU ellapsed time: 346294.187500 milliseconds

***************GPU Tiled Implementation***************
Calculated Output = 16384.000000
True Output = 16384
Output Difference = 0.000000

GPU ellapsed time: 42369.121094 milliseconds
```

Regular GPU and Tiled Implementation:

Matrices are created prior to matrix multiplication in the CPU and GPU

Regular GPU and Tiled Implementation			
Matrix Size	CPU Time [ms]	GPU Time [ms]	GPU Speed Gain
8192	4.32E+04	5.28E+03	8.181839393
16384	3.46E+05	4.24E+04	8.163370549
32768	2.77E+06	3.41E+05	8.135561422