Problem #1 Device Query

1. Suppose you are launching a one dimensional grid and block. If the hardware's maximum grid dimension is 65535 and the maximum block dimension is 512, what is the maximum number threads can be launched on the GPU?

   #blocks = 65535, #threads per block = 512

   Total Threads = 65535 * 512 = **33553920**

2. Under what conditions might a programmer choose not want to launch the maximum number of threads?

   The problem isn't parallelizable enough to completely saturate the hardware. The computation that needs to be done is global memory intensive and the memory bandwidth of the hardware is reached before the maximum number of threads is.

3. What can limit a program from launching the maximum number of threads on a GPU?

   There is not enough resources for another thread to use. These resources can be registers, shared memory, global memory.

4. What is shared memory?

   The memory that is local to a block (all threads within the block share access). Much faster than global memory.

5. What is global memory?

   Memory that is shared by all blocks - RAM that is local to the graphics card. Comparatively very slow to registers, shared and constant memory.

6. What is constant memory?

   Memory used to store data that will not change during the runtime of the kernel. Same locality as global memory, but is very fast as long as all threads access the same location.

7. What does warp size signify on a GPU?

   Each warp is a group of threads that run the same instruction at the same time. When talking about control divergence, it will only happen if threads within a warp diverge.

Problem #2 Vector Add

1. How many floating operations are being performed in your vector add kernel in terms of N, the size of the vector? Explain.

   N FLOPs. If each thread is computing "out[i] = in1[i] + in2[i];" then the only floating point operation is the addition, computing the index i along with using it is integer arithmetic. N threads are created to add N elements in the two vectors, so N FLOPs are performed.

2. How many global memory reads are being performed by your kernel in terms of N, the size of the vector? Explain.

   2N. Each thread reads an element from each input vector.
   Global memory reads in **bold:** out[i] = in1**[i]** + in2**[i]**

3. How many global memory writes are being performed by your kernel in terms of N, the size of the vector? Explain.

   N. Each thread writes a single element in the output vector.
   Global memory writes in **bold:** out**[i]** = in1[i] + in2[i]

4. Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

   If the instruction "in1[i] + in2[i]**"** isn't compiled into reads of in1[i] and then in2[i], then this could be explicitly done to take advantage of DRAM bursting.

   If the vectors are small enough, placing them into constant memory will provide smaller delay on the reads, but that isn't necessarily a kernel optimization.

5. Name three applications of vector addition.

   Many problems in classical physics add vectors together.

   Since sound waves are digitized as a vector, layering sounds is done through vector addition.

   Array operations

Problem #3 Basic Matrix Multiply

1. How many floating operations are being performed in your matrix multiply kernel in terms the following variables: numCRows, numCColumns, and numAColumns? Explain.

   2 FLOPs in **red**:

```
for (int i = 0; i < numAColumns; ++i)
{
        C[cRow + col] += A[aRow + i] * B[i * numBColumns + col];
}
```

   Each thread performs 2 * numAColumns FLOPs
   numberOfThreads = numCRows * numCColumns
   Total FLOPs = numberOfThreads * 2 * numAColumns
   Total FLOPs = numCRows * numCColumns * 2 * numAColumns

2. How many global memory reads are being performed by your kernel in terms of the following variables: numCRows, numCColumns, and numAColumns? Explain.

2 reads in **red:**

```
for (int i = 0; i < numAColumns; ++i)
{
        C[cRow + col] += A[aRow + i] * B[i * numBColumns + col];
}
```

Each thread performs 2 * numAColumns reads
numberOfThreads = numCRows * numCColumns
Total reads = numberOfThreads * 2 * numAColumns
Total reads = numCRows * numCColumns * 2 * numAColumns

3. How many global memory writes are being performed by your kernel in terms of the following variables: numCRows and numCColumns? Explain.

1 write in **red:**

```
for (int i = 0; i < numAColumns; ++i)
{
        C[cRow + col] += A[aRow + i] * B[i * numBColumns + col];
}
```

Each thread performs numAColumns writes
numberOfThreads = numCRows * numCColumns
Total writes = numberOfThreads * numAColumns
Total writes = numCRows * numCColumns * numAColumns

4. Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

Using a tiled approach where each thread in a block loads a single element in a "tile" of each of the input matrices into shared memory, and then a partial dot product is performed on the tile. This is then repeated until the full dot product is computed. This reduces the total number of global reads significantly by reusing the already loaded values stored in shared memory.

Making sure the loaded elements are within a DRAM burst.

5. Name three applications of matrix multiplication.

Graphics, Physics, AI

Problem #4 Tiled Matrix Multiply

1. How many floating operations are being performed in your matrix multiply kernel in terms the following variables: numCRows, numCColumns and numAColumns? Explain.

For my tile size of 32:
FLOPs in **red:**

```
const int numTiles = (numAColumns - 1) / TILE_SIZE + 1;
for (int i = 0; i < numTiles; ++i){
```

```
        ...
        for (int j = 0; j < TILE_SIZE; ++j){
            partialDotProduct += aTile[TILE_Y][j] * bTile[j][TILE_X];
        }
}
```

numTiles = (numAColumns - 1) / 32 + 1
Each thread performs 2 * 32 * numTiles FLOPS
numThreads = numCRows * numCColumns
Total FLOPS = 2 *32 * ((numAColumns - 1) / 32 + 1) * numCRows * numCColumns

2. How many global memory reads are being performed by your kernel in terms the following variables: numCRows, numCColumns, and numAColumns? explain.

```
for (int i = 0; i < numTiles; ++i){
        …
        aTile[TILE_Y][TILE_X] = A_TILE[TILE_Y * numAColumns + TILE_X];
        bTile[TILE_Y][TILE_X] = B_TILE[TILE_Y * numBColumns + TILE_X];
}
```

2 Reads per thread * number of tiles
numTiles = (numAColumns - 1) / 32 + 1
numThreads = numCRows * numCColumns
Total reads = ((numAColumns - 1) / 32 + 1) * numCRows * numCColumns

3. How many global memory writes are being performed by your kernel in terms of the following variables: numCRows and numCColumns? Explain.

   The only write being performed is at the end of the kernel when writing the final dot product to the C array, which is 1 per thread.
   Total Writes = numThreads = numCRows*numCColumns

4. Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

   Change the order of the elements in memory in the B matrix from row major to column major. With this change, the memory accesses in the matrix as the tile moves down the columns would be coalesced and make use of optimal DRAM bursting.

5. Compare the implementation difficulty of this kernel compared to the BasicMatrixMultiply problem. What are the new code additions that programmers can make errors with this implementation?

There are more edge cases that need to be handled. In the basic matrix multiplication, threads that are out of bounds of the matrix can simply return, in this approach, these threads must still participate in the loading of elements.

Visualizing the tiles moving across the matrices and being able to translate what each process in the tile needs to be doing into code could be difficult. In the basic matrix multiplication, it's a single for loop, while in this it's two.

Getting the indexing into the arrays correct is more difficult because the "start" of the tile is always moving.

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

   Mimic the approach to using shared memory, compute the matrix in tiles. Load the matrix into global memory (assuming enough RAM), then from the host launch kernels in serial that compute the elements of a smaller portion of C.

7. Suppose you have matrices that would not fit in global memory. Describe an approach that would perform matrix multiplication in this case.

   Mimic the approach to using shared memory, compute the matrix in tiles. Load a set of full Rows from Matrix A and a set of full Columns from Matrix B into global memory (as much as the RAM of the card will allow), then compute the elements in that tile of C. Which could take the same tiled approach using shared memory. After that tile is completed, load the next set of columns, reusing the rows and compute the next tile.