Question 3 - Tiled Matrix multiplication

(1) How many floating-point operations are being performed in your matrix multiply kernel in terms of numCRows, numCColumns, and numAColumns?

In matrix multiplication, the number of floating-point operations (FLOPs) for computing each element in the result matrix C is determined by the number of columns in matrix A, which is also the number of rows in matrix B (denoted by numAColumns). For each element C[i][j], we compute a dot product between the i-th row of A and the j-th column of B.

- To compute each element C[i][j], there are numAColumns multiplications and numAColumns - 1 additions (since additions are between intermediate results of multiplications). Hence, each element requires 2 * numAColumns floating-point operations (FLOPs).
- The total number of elements in C is numCRows * numCColumns.

Thus, the total number of floating-point operations is:Total FLOPs = numCRows *numCColumns * 2 * numAColumns

(2) How many global memory reads are being performed by your kernel in terms of numCRows, numCColumns, and numAColumns?

- For each tile, every thread block reads from the global memory into shared memory.
- The number of reads from matrix A depends on the number of rows in A (numARows) and the number of columns in A (numAColumns). However, we are dividing A into tiles, and each tile corresponds to a block of size TILE_WIDTH × TILE_WIDTH. Each thread reads one element of A from global memory into shared memory for each tile. For matrix B, the logic is similar.

- If you break A and B into tiles, for each tile of A and B, the number of global memory reads is:
Reads per tile = TILE_WIDTH^2

Each thread reads one element of `A` and one element of `B` per iteration. The number of iterations (tiles) is numAColumn/TILE_WIDTH

Thus, the total number of global memory reads is proportional to:
2*(numAColumn/TILE_WIDTH) * TILE_WIDTH^2 *number of tiles
This is the approximate number of global memory reads.

(3) How many global memory writes are being performed by your kernel in terms of numCRows and numCColumns?

Each element in the result matrix C is written once to global memory after its computation in shared memory. Hence, the number of global memory writes is equal to the number of elements in matrix C, which is:
Total writes = numCRows * numCColumns

(4) What further optimizations can be implemented to your kernel to achieve a performance speedup?

Several optimizations can improve performance:
1. Coalesced Memory Access: Ensure that global memory accesses are coalesced, meaning threads in the same warp access consecutive memory locations. This minimizes memory transaction overhead.
2. Memory Bank Conflicts: Reduce shared memory bank conflicts by ensuring that multiple threads do not access the same memory bank simultaneously.
3. Instruction-Level Parallelism: Increase instruction-level parallelism by overlapping memory loads and computations. This can be achieved by preloading the next tile while computing the current one.
4. Use of Registers: Leverage registers to store intermediate results and reduce the pressure on shared memory.
5. Asynchronous Memory Copy: Use CUDA streams to overlap memory transfers with computations to reduce overall execution time.

(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?

The tiled matrix multiplication using shared memory is more complex than the basic matrix multiplication due to:
- Shared Memory Management: The programmer must manually handle shared memory, ensuring tiles are loaded, synchronized, and computed correctly. Errors can arise from incorrect indexing, failure to pad matrices properly, or failure to synchronize threads __syncthreads().
- Thread Block and Grid Dimensions: Proper handling of thread block and grid dimensions is essential, especially when matrix sizes are not divisible by TILE_WIDTH. Misalignment can cause boundary errors.
- Synchronization Errors: Misuse of __syncthreads() can lead to race conditions or deadlocks if threads do not synchronize correctly.

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

If matrix dimensions exceed the maximum allowable thread block size, you can divide the matrix into submatrices and perform matrix multiplication in chunks. This technique is called tiling or chunking:
- Split the large matrices into smaller tiles that fit within the allowed thread block size.
- Perform matrix multiplication on each tile independently.
- Accumulate the results in a global memory buffer to form the final product.

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

When matrices are too large to fit into global memory, the following approaches can be used:
1. Out-of-Core Computation : Split the matrices into blocks that fit into global memory. Load the blocks into memory sequentially, perform partial multiplications, and accumulate the results.
2. Streaming : Use CUDA streams to overlap computation with memory transfers. Load parts of the matrices into memory while processing other parts to hide memory latency.
3. Hierarchical Decomposition : Break down the matrix multiplication into smaller sub-problems that fit into the available memory and use iterative approaches to compute the result block by block.