Question 2 - Basic Matrix Multiplication

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

In matrix multiplication, each element C[i][j] of the resulting matrix is computed by performing a dot product of the i-th row of matrix \(A\) with the j-th column of matrix B. The number of floating point operations (FLOPs) required for each element of the result matrix C is equal to the number of elements in the row of A or the column of B, which is numAColumns.

Each multiplication of two numbers and the subsequent addition is considered two floating point operations (one for multiplication and one for addition).

Therefore, for each element in matrix C, there are (2 * numAColumns) floating operations (since you perform numAColumns multiplications and numAColumns-1 additions, resulting in approximately 2 * numAColumns FLOPs per element in C ).

For the entire matrix C, which has numCRows * numCColumns elements, the total number of floating point operations is:

Total FLOPs = 2 * numCRows * numCColumns * numAColumns

This is the total number of floating point operations performed by the matrix multiplication kernel.

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

To compute each element of matrix C[i][j], the kernel must read the entire i-th row of matrix A and the entire j-th column of matrix B. The number of reads per element of matrix C is numAColumns from A and numAColumns from B, resulting in:

Reads per element of C = numAColumns + numAColumns = 2 * numAColumns

For the entire matrix C, which has numCRows * numCColumns elements, the total number of reads is:

Total reads = 2 * numAColumns * numCRows * numCColumns

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

The result of each dot product is written to the corresponding position in the result matrix C. Since each element in matrix C requires exactly one write, the total number of global memory writes is simply equal to the number of elements in matrix C, which is:

Total writes = numCRows * numCColumns

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

Here are a few optimizations that could improve the performance of the matrix multiplication kernel:

1. Tiling with Shared Memory: Instead of loading individual elements of matrices A and B directly from global memory for each thread, you can load sub-blocks (tiles) of the matrices into shared memory. This would significantly reduce the number of global memory accesses, as the same data would be reused by multiple threads within a block.

2. Coalesced Memory Access: Ensuring that global memory accesses are coalesced, meaning that consecutive threads in a block access consecutive memory addresses, will improve memory bandwidth and reduce access time.

3. Loop Unrolling: Manually unrolling loops can reduce the number of loop control instructions and can help the compiler better optimize the generated code.

4. Occupancy Maximization: By adjusting block and grid sizes, you can maximize the occupancy (i.e., the number of active warps running on the GPU), leading to better utilization of the hardware.

5. Use of Registers: Storing intermediate results in registers rather than repeatedly accessing global memory can reduce memory access latency.

(5) Name three applications of matrix multiplication.

1. Graphics Rendering: Matrix multiplication is used extensively in computer graphics for transformations such as scaling, rotating, and translating objects in 3D space.

2. Neural Networks: In deep learning, matrix multiplication is a fundamental operation in the forward and backward propagation of data through neural networks.

3. Physics Simulations: Matrix multiplication is employed in simulations of physical systems, such as the multiplication of state vectors by transformation matrices to model motion, deformation, or forces.