First question - Vector Add

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

In the CUDA kernel vecAdd, each thread performs one addition operation for each element of the vectors. Since there are N elements in each vector, and each element is added to the corresponding element in the other vector, the number of floating-point operations is directly proportional to the size of the vector N

Thus, N-floating-point addition operations are performed in total, one for each element of the vector.

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

For each element of the output vector, the kernel reads two elements from global memory: one from the first input vector ( in1 ) and one from the second input vector ( in2 ). Therefore, the total number of global memory reads is:

Number of reads = 2 \* N

So, 2N global memory reads are being performed, with one read from each of the input vectors.

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

For each element of the result vector, the kernel writes one element to global memory. Since there are N elements in the output vector ( out ), the number of global memory writes is simply N .

Thus, N global memory writes are performed.

4. Name three applications of vector addition.

1. Image Processing: In image blending or compositing, vector addition can be used to combine pixel values of two images. Each pixel can be represented as a vector (RGB channels), and addition can blend or mix the pixel colors.

2. Physics Simulations: In physical simulations, vector addition is used to compute the resultant force, velocity, or position of an object by adding forces, velocities, or displacements, respectively.

3. Machine Learning: In the training of neural networks, vector addition is used during backpropagation to update weights by adding gradients to the current weights. It's also used to sum weighted inputs at the nodes of the network.

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.

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.