Problem statement

PA2: Optimizing matrix multiplication in CUDA

In this assignment, you will implement matrix multiplication in CUDA and optimize it using several techniques.

Before you begin

Familiarize yourself with the CUDA programming framework from the CUDA programming guide and several CUDA tutorials provided online like this basic cuda tutorial. You can practice writing CUDA code on leetgpu.com playground, which runs CUDA code on a GPU simulator. To get performance numbers on a real GPU, you can use a Kaggle notebook with GPU acceleration turned on.

Part A: Matmul on CPU vs GPU

In this part, you will write code for multiplying two N × N matrices (C = A × B) (float datatype), in the following four different ways.

1. CPU implementation: Use a simple, naive triple-loop matrix multiplication on the CPU (with O3 flag). Do not use tiling, unrolling, SIMD or any other optimizations.
2. GPU implementation (Naive Kernel): Write a simple CUDA kernel implementing the naive matmul algorithm, without coalesced access for threads in a warp. Use 32 × 32 threads per block. Increase the number of blocks as N grows.
3. GPU implementation (Coalesced Kernel): Improve upon the previous CUDA kernel to ensure that threads in a warp access contiguous elements in a matrix, leading to coalesced accesses.
4. GPU implementation (Optimized cuBLAS Kernel): Use the cuBLAS library's optimized GEMM kernel (e.g., cublasSgemm or cublasDgemm) to perform matrix multiplication. Note that this library expects column-major matrices, while hosts use row-major format, so you must suitably transpose the arguments.

For each of the above variants, vary the matrix size N from 1024 to 32K, with a multiplicative step size of 2. (You may stop using the CPU implementation after a point if it gets too slow.) Plot the following three graphs, with N on the X-axis, and the metrics described below on the y-axis, with different lines for the implementation variants described above.

1. Total execution time of the matmul program, as measured by a command like `time`.
2. Kernel execution time for the GPU programs, measured by tools like `nvprof` or `ncu`. Note that the kernel execution time only includes the time on the GPU, and does not consider additional overheads like CPU setup time and data transfer time.
3. Achieved TFLOP/sec, computed by dividing the FLOPs in each matmul (2N^3) by the time taken (kernel execution time). This is similar to a roofline plot, only the x-axis is N and not arithmetic intensity. You must compute the peak TFLOP/sec of your GPU from the GPU specifications, and have this number handy to compare with your achieved TFLOP/sec for the various implementations.

What conclusions can you draw from these graphs above? Up to what values of N does a CPU implementation look feasible? What are the performance benefits of coalesced access? How does your naive GPU matmul compare with the cuBLAS optimized matmul? Are you able to observe the expected shape of the roofline plot?

Part B: Matmul with tiling

Next, you will optimize your coalesced access matmul kernel with tiling using shared memory. You can then plot the same graphs as earlier for your optimized implementations, and see how closely you can match the optimized cuBLAS implementation with the tiling optimization.

You must implement the following ways of tiling.

1. Every thread in a block loads one element each of A and B, and computes one element of C. This simple way of tiling has already been discussed in class.
2. Every thread loads one element of A, multiple elements of a row of B in a strided manner, and computes multiple elements of C. More on this here.
3. Every thread loads multiple contiguous values of A, and one value of B, to compute multiple elements of C. More on this here.

As you can see, there are many ways of performing tiling, and we have only listed a few sample variants for you to practice with. Please feel free to experiment with other tiling variants as well. For every variant, you can also vary the tile size and observe its impact on the various performance metrics like execution time.

Part C: More optimizations to matmul

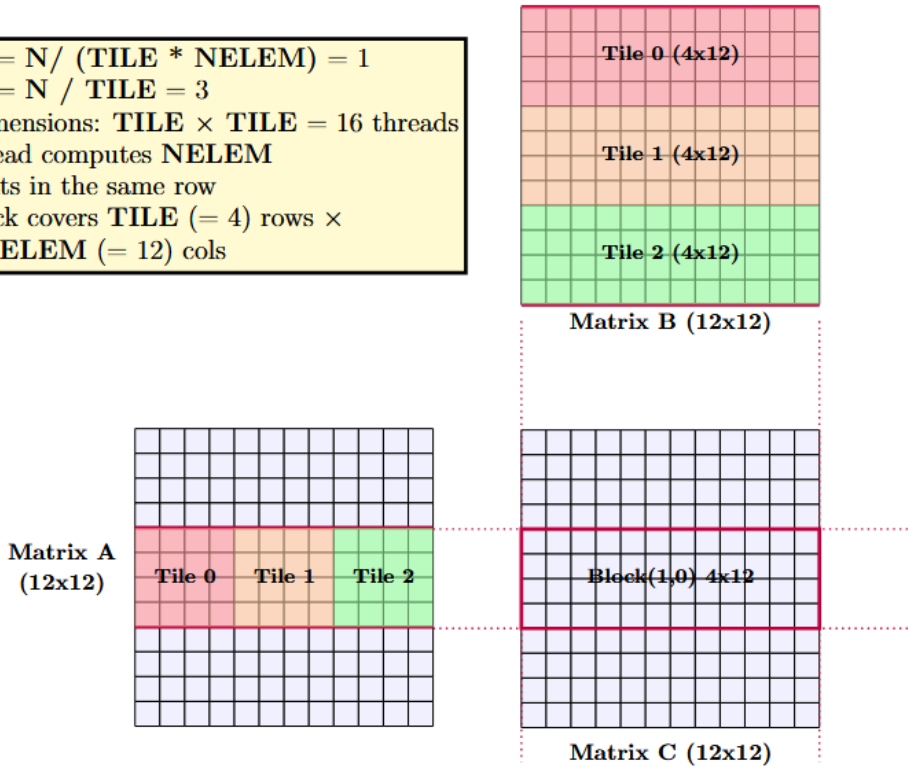
There are many other optimizations one can do to matrix multiplication to match the performance of optimized CUDA libraries. For example, you can minimize shared memory usage by loading one of the two matrices (A and B) into registers (i.e., into regular local variables, which will use registers), and using shared memory for the other. Many more such optimizations are described in several blog posts online like this article. Feel free to try out some of these ideas, which may require some usage of advanced CUDA APIs as well. The goal is to eventually match the performance of optimized matmul libraries, and reach close to the GPU's peak performance.

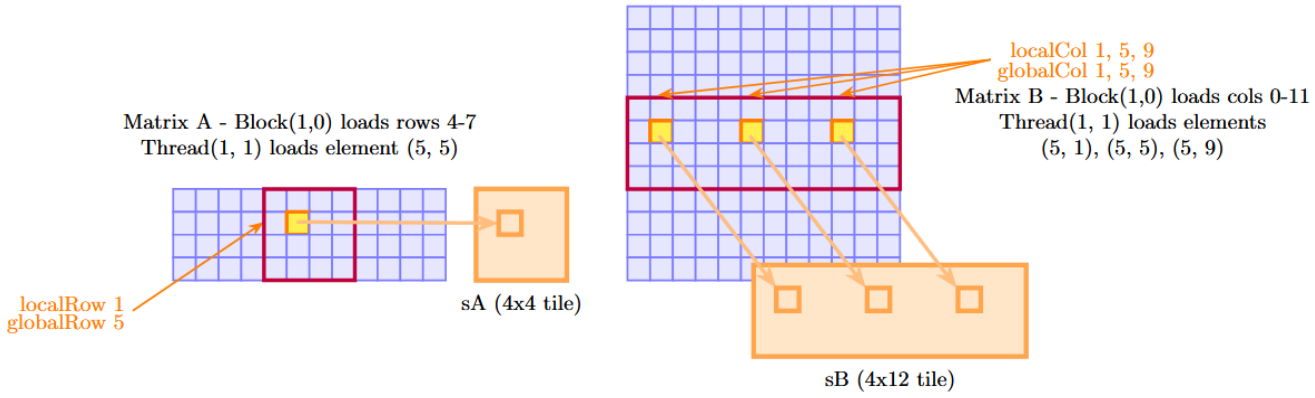
Row-based 1D tiling

Tiled Matrix Multiplication (Row-per-Thread)

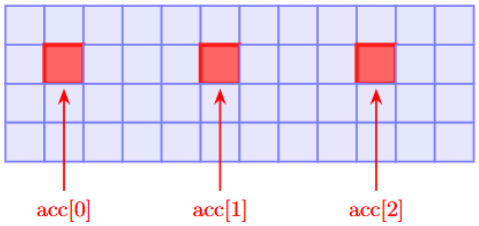
N = 12, TILE = 4, NELEM = 3

- blocks_x = **N/ (TILE * NELEM)** = 1
- blocks_y = **N / TILE** = 3
- Block dimensions: **TILE** × **TILE** = 16 threads
- Each thread computes **NELEM** = 3 elements in the same row
- Each block covers **TILE** (= 4) rows × **TILE * NELEM** (= 12) cols





Matrix C - Block(1,0) writes C[4:8][0:12]
Thread(1, 1) writes elements (5, 1), (5, 5), and (5, 9)



Thread(1,1) in Block(1,0):

Position:
- blockRow = 1, blockCol = 0
- localRow = 1, localCol = 1
- globalRow = blockRow * TILE + localRow = 5
- baseGlobalCol = blockCol * (TILE * NELEM) = 0

For each tile t = 0, 1, 2:
- aCol = t * TILE + localCol
- bRow = t * TILE + localRow
- **Loads:** A[globalRow][aCol] → sA[localRow][localCol]
- **Loads:** for e = 0, 1, 2:
  gCol = baseGlobalCol + localCol + e * TILE
  B[bRow][gCol] → sB[localRow][localCol + e * TILE]
- Sync threads
- **Compute:** for k = 0 to TILE-1, for e = 0 to NELEM-1:
  acc[e] += sA[localRow][k] * sB[k][localCol + e * TILE]
- Sync threads

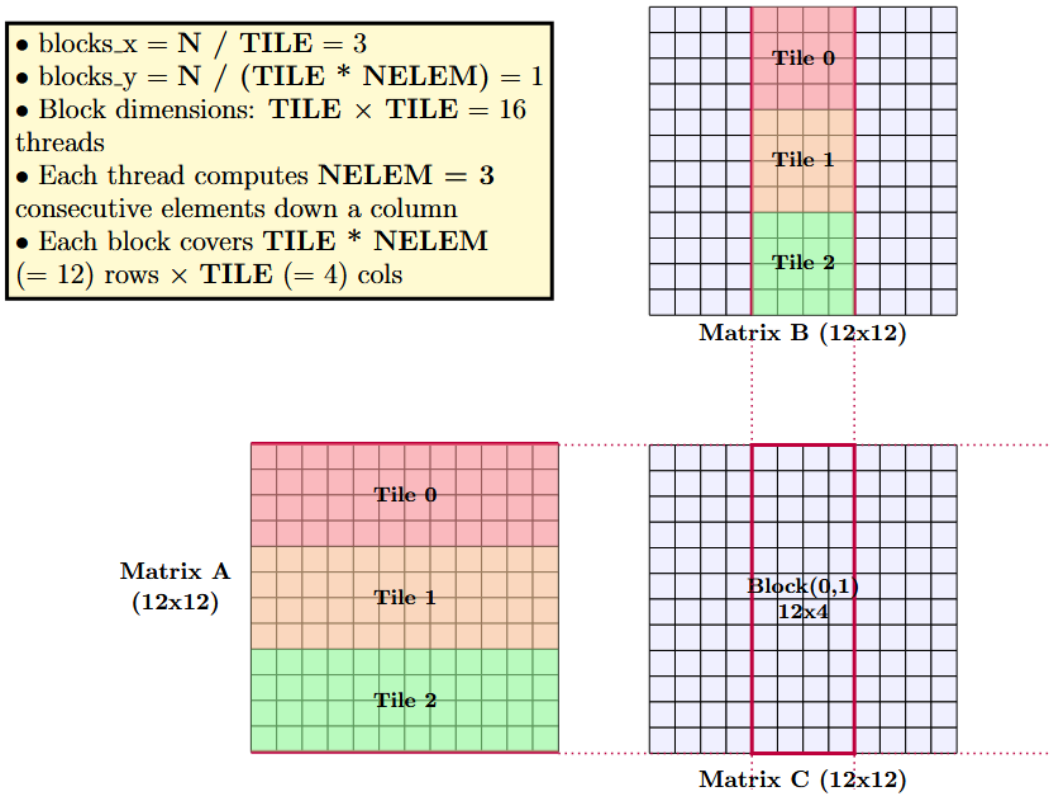
Final writes: for e = 0, 1, 2:
- gcol = baseGlobalCol + localCol + e * TILE
- C[globalRow][gcol] = acc[e]

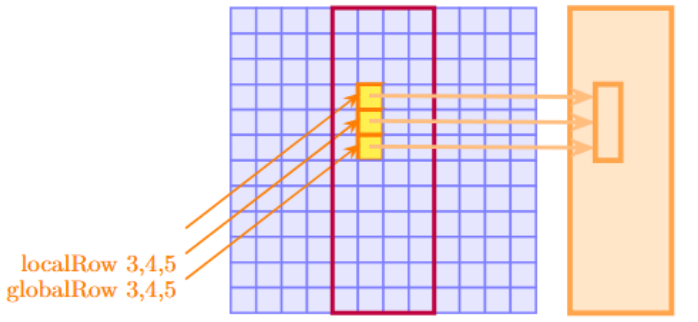
Column-based 1D tiling

Tiled Matrix Multiplication (Column-per-Thread Contiguous)

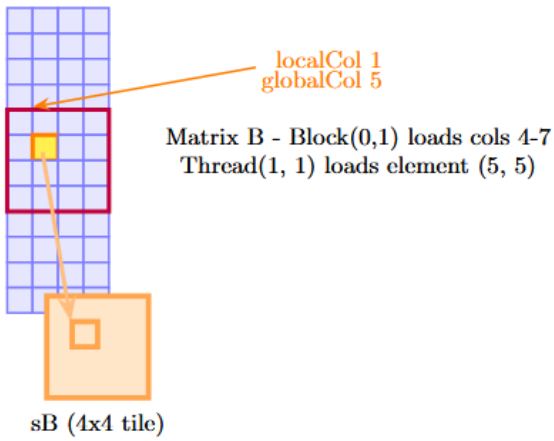
N = 12, TILE = 4, NELEM = 3

- blocks_x = **N** / **TILE** = 3
- blocks_y = **N** / (**TILE** * **NELEM**) = 1
- Block dimensions: **TILE** × **TILE** = 16 threads
- Each thread computes **NELEM** = **3** consecutive elements down a column
- Each block covers **TILE** * **NELEM** (= 12) rows × **TILE** (= 4) cols





Matrix A - Block(0,1) loads rows 0-11
Thread(1, 1) loads consecutive elements
(3, 5), (4, 5), (5, 5)



Matrix B - Block(0,1) loads cols 4-7
Thread(1, 1) loads element (5, 5)