CS 610 Coding Assignment 2

Winter 2026

Total points: 10

Learning Outcomes

- Understand how to write a CUDA program from scratch
- Understand how to use shared memory to improve performance
- Understand how to use tiling in shared memory for matrix multiply
- Understand how to design a computing and data copy pipeline

For tasks 1&2:

- You may develop and debug your code using any GPU, but you should run performance tests and report the final performance you achieve on A100.
- To make debugging easier, you may choose smaller matrix sizes.
- For a performance test on the GPU cluster server, choose (m=k=n=10000) as the input sizes. *You need to adjust the host memory limit to fit the input matrices when submitting your job to Talapas.*
- Submit task1. cu, task2.cu, and a report (Word/PDF) showing the time and performance you got for each task.

Task 1 (4 points)

You need to start with a new empty source code file named: task1.cu

Task 1 requires that we develop a matrix-matrix multiplication kernel that computes:

$$C = A \cdot B$$

A (input), B (input), and C (output) are matrices. The dimensions of the matrices are A (m*k), B(k*n), and C (m*n).

1) **(0.5 point)** Allocate and initialize all matrices with random single-precision floating point (i.e., float type) numbers in the range (0-1)
2) **(1 point)** Design a kernel that uses total m*n threads to compute total m*n elements in the output matrix C.
3) **(0.5 point)** Try the following thread block sizes (8*8), (16*16), and (32, 32) and select the fastest configuration.
4) **(0.5 point)** Time your kernel execution for input size (m=k=n=10000)
5) (0.5 point) Compute the GFlops performance of your kernel using: m*k*n*2 / 1e9 / <exec time in second>
6) **(0.5 point)** Your kernel should work with any valid input sizes. This means you need to handle cases where the dimensions are not exact multiples of the thread block sizes.
7) **(0.5 point)** Implement a CPU version of the matrix-matrix multiplication function to verify your results

Task 2 (6 points)

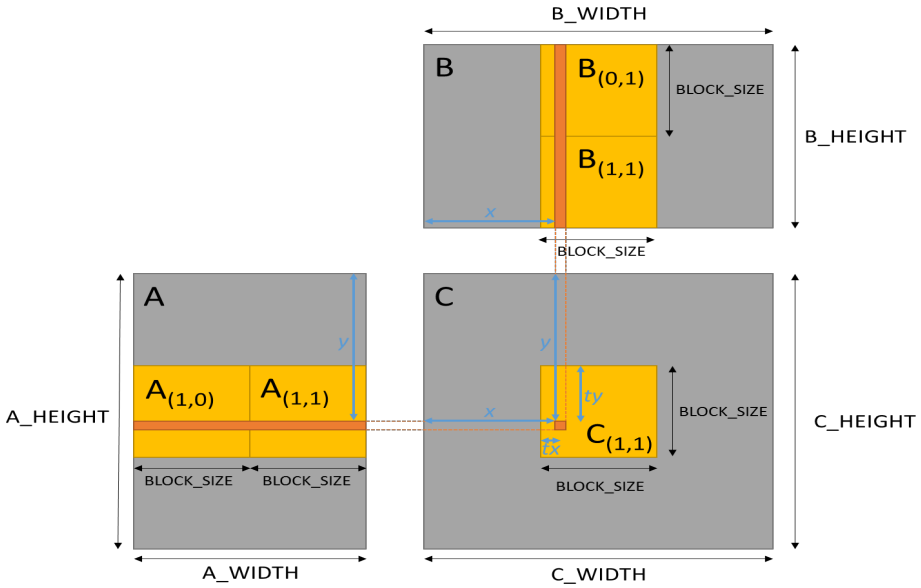

Figure 1 - Blocked matrix multiply (See text for description). Note: indices of sub-block are given as (y, x)

You need to start with a new empty source code file named: task2.cu.

For task 2, we are going to optimize the matrix-matrix multiplication kernel using shared memory based on the code of Task 1. The implementation is currently very inefficient, as it performs `A_WIDTH × B_HEIGHT` memory loads to compute each product (i.e., the value in matrix C). To improve this, we will implement a blocked matrix multiply that uses CUDA's shared memory to reduce the number of memory reads by a factor of `BLOCK_SIZE`.

1) **(4 points)** Implementing a blocked matrix multiply will require iteratively loading square sub-matrices of matrices A and B into shared memory so we can compute the intermediate results of the sub-matrix products.

   In the example Figure 1 (above), the sub-matrix $C_{(1,1)}$ can be calculated by a square thread block of `BLOCK_SIZE` by `BLOCK_SIZE`, where each thread (with location `tx`, `ty` in the square thread block) performs the following steps, which require two stages of loading matrix tiles into shared memory.
   a. Load the two sub-matrices $A_{(1,0)}$ and $B_{(0,1)}$ into shared memory. For these submatrices each thread should
      i. Load an element of the sub-matrix $A_{(1,0)}$ into shared memory from the matrix A at position `(ty+BLOCK_SIZE, tx)`
      ii. Each thread should load an element of the sub-matrix $B_{(0,1)}$ into shared memory from the matrix B at position `(ty, tx+BLOCK_SIZE)`
      iii. Synchronize to ensure all threads have completed loading sub-matrix values to shared memory
   b. Compute the dot product of each row in sub-matrix $A_{(1,0)}$ with each column in the sub-matrix $B_{(0,1)}$, storing the result in a local variable. Achieved through the following steps.
      i. Iterate from `0` to `BLOCK_SIZE` to multiply row `ty` of $A_{(1,0)}$ (from shared memory) by column `tx` of $B_{(0,1)}$ (from shared memory) to calculate the sub-matrix product value.
   c. Store the sub-matrix product value in a thread-local variable.
      i. Synchronize to ensure that all threads have finished computation
   d. Repeat steps 1 & 2 for the next sub-matrix (or matrices in the general case), adding each new dot product result to the previous result. For the example in the figure, only one more iteration is required to load the final 2 submatrices. E.g.
      i. Load an element of the sub matrix $A_{(1,1)}$ into shared memory from Matrix A at

position (`ty+BLOCK_SIZE`, `tx+BLOCK_SIZE`)

ii. Load an element of the sub matrix $B_{(1,1)}$ into shared memory from matrix B at position (`ty+BLOCK_SIZE tx+BLOCK_SIZE`,)

iii. Synchronize to ensure all threads have completed loading sub-matrix values to shared memory

iv. Iterate from `0` to `BLOCK_SIZE` to multiply row `tx` of $A_{(1,1)}$ (from shared memory) by column ty of $B_{(1,1)}$ (from shared memory) to calculate the sub-matrix product value.

v. Add this sub-matrix product value to the one calculated for the previous sub-matrices.

e. Store the sum of the sub-matrix dot products into global memory at location `x`, `y` of Matrix C.

2) **(2 points)** Same as Task 1, you should find the best block size, report time, and performance you achieved, and use CPU matrix-matrix multiplication to verify correctness