Assignment #2

CS300/CSD2170 Programming Massively Parallel Processors, Fall 2022

Due Date:	As specified on the moodle
Topics covered:	Tiled Matrix Computation, Shared Memory
Deliverables:	The submitted project files are the CUDA C/C++ source code (kernel.cu). The file should be put in a folder and subsequently zipped according to the stipulations set out in the course syllabus.
Objectives:	Learn how to use shared memory to write more efficient kernel functions.

Programming Statement

This is a CUDA C programming assignment. Students are expected to finish the programming for tiled Matrix Multiplication.

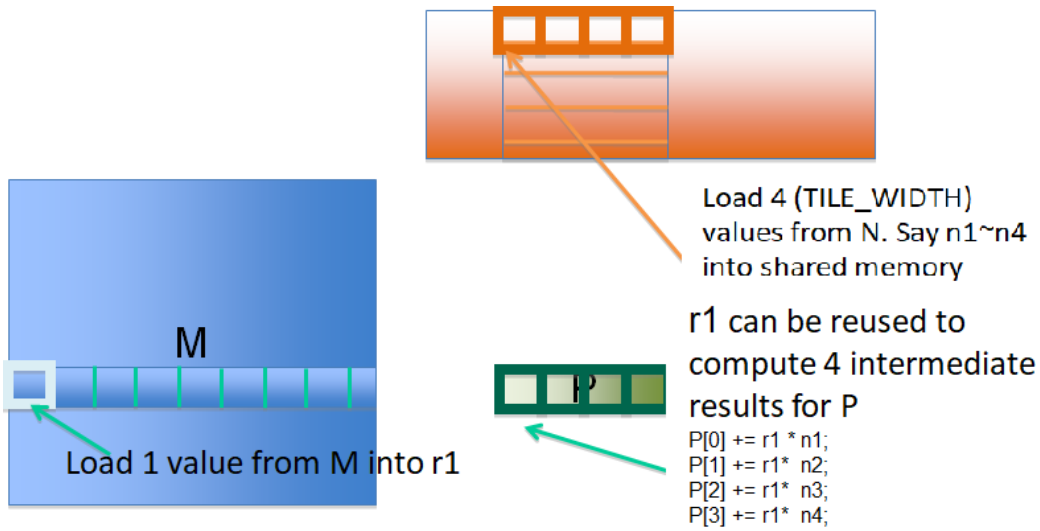
Problem Statement

In this assignment you are required to implement a floating point matrix multiplication program using CUDA C/C++ and study the effect of block size and thread size on performance. You can consult the sample CUDA code and lecture notes provided in the class.

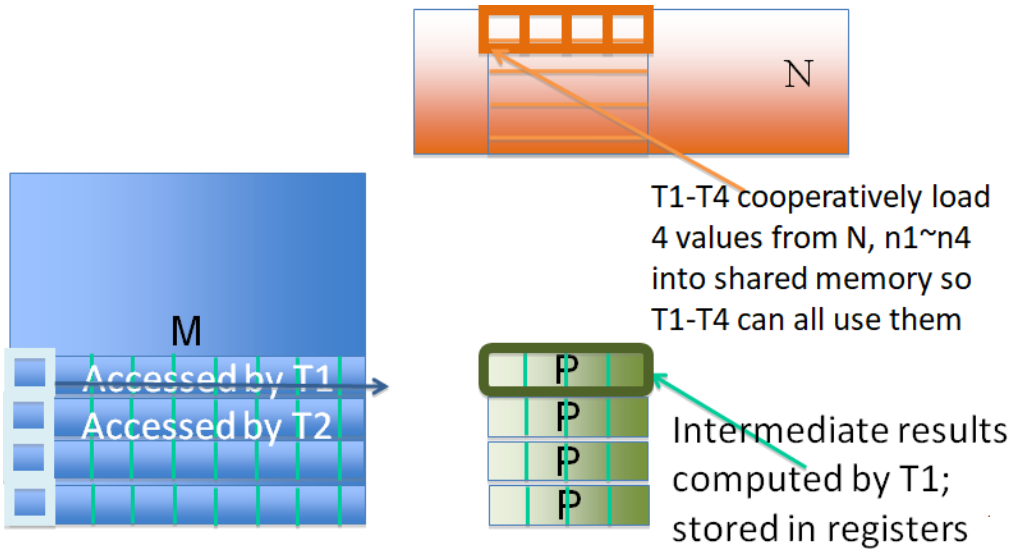
Assume matrix multiplication of M and N produces the result P. In the tiling method for matrix multiplication, `__synchthreads()` is used in the inner loop of each thread. In each iteration,

1. only a subset of threads load M and N elements
2. Call `__synchthreads()`
3. All threads calculate one step of the inner product
4. Call `__synchthreads()`
5. Go to the next iteration

In this assignment, we implement a new tiling method. As shown below,



each thread calculates a horizontal subset of P elements and data loaded from M can be reused through registers, namely, register tiling. Multiple threads collaborate to load N elements into shared memory as follows:



In one iteration, each thread loads one M element into register, accesses N elements from shared memory and calculates one step for P elements.

In one iteration, each block loads $TILE\_WIDTH\_M$ M elements into registers, loads $TILE\_WIDTH\_N \times TILE\_WIDTH\_RATIO\_K$ N elements into shared memory, where $TILE\_WIDTH\_M$ is the number of threads in thread block (e.g. 64 or more), $TILE\_WIDTH\_N$ is the number of threads folded into one thread (e.g. 16 or more), and

$TILE\_WIDTH\_RATIO\_K = \frac{TILE\_WIDTH\_M}{TILE\_WIDTH\_N}$.

Instead, all threads in a block collaborate to load a tile of $TILE\_WIDTH\_N \times TILE\_WIDTH\_RATIO\_K$ N elements into shared memory. Every thread loads one N element so there is no divergence. Each thread loads $TILE\_WIDTH\_RATIO\_K$ M elements into registers. Each thread calculates $TILE\_WIDTH\_RATIO\_K$ steps for $TILE\_WIDTH\_N$ P elements.

The steps for the algorithm:

1. Declare shared memory array for N elements with the size of $TILE\_WIDTH\_RATIO\_K \times TILE\_WIDTH\_N$.

2. Declare output array variable for P elements with the size of $TILE\_WIDTH\_N$ and initialize the output array variable.

3. Loop over the input tiles (the number of iterations $= \frac{k-1}{TILE\_WIDTH\_RATIO\_K} + 1$, where k the number of the columns of matrix M.)

   - Load the tile of N (size $= TILE\_WIDTH\_RATIO\_K \times TILE\_WIDTH\_N$) into shared memory. Note that one block has $TILE\_WIDTH\_M$ threads, each loading one N element into shared memory.
   - Loop over elements inside the tile of N (the number of iteration $= TILE\_WIDTH\_RATIO\_K$).
     - Load tile of matrix M into register (i.e. each thread load one M element into the local variable in this iteration)
     - Loop over and update the output elements in the output array variable assigned to this thread. Note that output array variable is local variable. It accumulates the partial results. In this innerloop, the number of iteration is $TILE\_WIDTH\_N$.

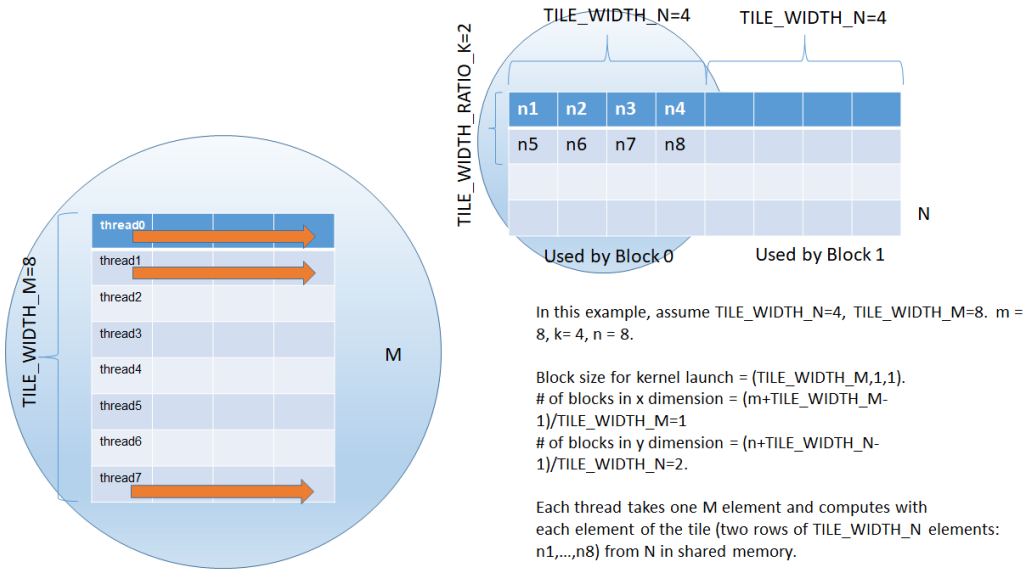
4. Store the output array variable to P elements (each thread stores $TILE\_WIDTH\_N$ P elements and one block outputs $TILE\_WIDTH\_N \times TILE\_WIDTH\_M$ P elements).

When you launch the kernel, the grid and block size should be respectively :

```
dim3 dimGrid((m+TILE_WIDTH_M-1)/TILE_WIDTH_M, (n+TILE_WIDTH_N-1)/TILE_WIDTH_N);
dim3 dimBlock(TILE_WIDTH_M, 1);
matrixMultiply<<<dimGrid, dimBlock>>>(P, M, N, m, n, k);
```

where $m \times k$ is the size of matrix M and $k \times n$ is the size of matrix N. In the kernel function, shared memory size should be declared something like

```
float sdata[TILE_WIDTH_RATIO_K][TILE_WIDTH_N];
```



`Please note that before the kernel function to perform matrix multiplication is launched, the input matrix M should be in column-major format while the input matrix N should be in row-major format. The resulting matrix P from the kernel is in column-major format.`

You are required to turn in your homework that is compilable under the Windows environment using Visual Studio in our lab.

Grading Guideline

Please ensure that:

1. Functional Correctness: Produces correct result output for test inputs.
2. Coding:
   (a) Correct usage of CUDA library calls and C extensions.
   (b) Correct usage of thread id's in matrix computation.

Your submission will be graded according to the test results. If your code fail to pass the test cases, zero mark will be given. The latency could be tested for different matrix size up to e.g. $2,000$.