CUDA Lab 5 Matrix multiplication in CUDA

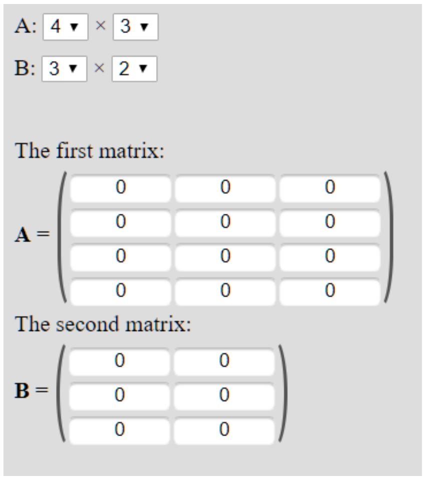
Matrix multiplication is a fundamental task in science and engineering as well artificial neural network. Matrix multiplication algorithms exhibit similar optimization techniques for many other algorithms, that is why matrix multiplication is often used as one of the most popular examples in learning parallel programming. The main objectives to be achieve in this lab are:

1. Understand the basic rules of performing matrix multiplication
2. Learn how to implement matrix multiplication in CUDA using one single block of threads
3. Identify the performance issues concerning one-thread block based matrix multiplication
4. Understand how to decompose the matrix multiplication of two large matrices into a set of smaller submatrix multiplications
5. Learn how to implement a 2D grid of thread block based matrix multiplication.

Exercise 1. A simple matrix multiplication program in CUDA using one thread block

To complete this task, you need to have a basic understanding of the matrix multiplication problem. Please let us know if you have any question about it. The best starting point of doing the task is to use the default CUDA sample created directly in VS 2017.

1. Start from simple matrices and work on paper how to multiply two matrices. It may be helpful to use an online matrix multiplication calculator, for instance, https://onlinemschool.com/math/assistance/matrix/multiply/, to define some simple matrices A and B, and to test and check the accuracy of your implementation.
2. Find the sizes of matrices A and B and define correspondingly the array sizes required for storing the elements of matrices A and B. For example, if the shapes of your matrices are



A: 4 × 3

B: 3 × 2

The first matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The second matrix:

$$\mathbf{B} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Yyou can define the sizes of your matrices in the following way:

```
    const int heightA = 4;
    const int widthA = 3;

    const int heightB = 3;
    const int widthB = 2;

    const int arraySizeA = heightA * widthA;
    const int arraySizeB = heightB * widthB;
    const int arraySizeC = heightA * widthB;
```

3. First, implement a solution in C++ to the problem (host device only) to make sure you understand how to perform matrix multiplication.
4. To implement a CUDA solution, let us first consider a one-thread block based solution. For the shapes of the matrices A and B shown above, matrix AxB has a size of `heightA x widthB`. Thus, we can define the two simple matrices by modifying the default arrays a[ ] and b[ ] and store the matrices in a[] and b[] respectively in a row by row manner:

```
const int a[arraySizeA] = { … … };
const int b[arraySizeB] = { … … };
int [arraySizeC] = { 0 };
```

5. Go to the kernel addKernel() and modify it to calculate array c[ ] based on the rule of matrix multiplication. There are different ways of computing each element of c[ ], which stores the result matrix AxB. The method introduced in the lecture is directly based on array a[ ] and b[ ]. Another way is to create two 2D arrays, say, d_A[][] and d_B[][], in shared memory corresponding to the original shapes of A, B. Then the matrix multiplication can be implemented in CUDA in the same way as it is implemented in C++.

Exercise 2. Compare the performance of the CUDA solution against the CPU solution

Consider the computation times taking for CPU and CUDA solutions corresponding to different sizes of matrices. To put things simple, we can just consider squared matrices. For instance, both A and B are both square matrices and of the same size, for instance, 8x8, 32 x 32, 256 x 256, 512 x 512, 1024x1024. Record the computation times used by your CUDA solution and your C++ solution corresponding to each of the cases.

Exercise 3. Matrix multiplication in CUDA using multiple thread blocks (optional)

1. Identify the bottleneck of your solution and consider how to improve the performance of you CUDA solution by using multiple thread blocks.
2. You may need to understand the theory of submatrix-based matrix multiplication to implement a multiple blocks of thread based CUDA solution.