# Lab 3: Tiled Matrix Multiplication

## 1. Objective

The purpose of this lab is to get you familiar with using shared memory to write optimized kernel algorithms by implementing a “tiled” version of matrix multiplication.

## 2. Procedure

**Step 1:** Download the starter code from the GitHub classroom repo.

**Step 2:** Edit <lab-directory>/main.cu and <lab-directory>/kernel.cu to include the host setup code and device kernel code where indicated by the //@@ comment lines. Edit the Makefile to build your project.

**Step 3:** Compile and test your code against each of the datasets provided. You can launch your program using commands of this structure:

./program -e <expected\_output\_file> -i <input\_file\_1>,<input\_file\_2> -o <output\_file> -t <type>

The files input\_file\_1 and input\_file\_2 define the two input matrices to be multiplied. The file expected\_output\_file is what the result of the multiplication should be. There should not be any whitespace on either side of the ','. For example, here is a test against the first data set.

./lab3 -e data/0/output.raw -i data/0/input0.raw,data/0/input1.raw -o myoutput -t matrix

Your code is expected to work for varying input dimensions (indeed, the various data sets do this) – which may or may not be divisible by your tile size. It is a good idea to test and debug initially with examples where the matrix size is divisible by the tile size, and then try the boundary cases.

**Step 5:** Answer the following questions in a new file named <lab-directory>/answers.txt: Where needed, assume you've launched your program to multiply two 64x64 matrices.

1. How many floating-point operations are being performed by all the threads in the grid as a result of launching your matrix multiply kernel?
2. How many global memory reads and writes are being performed by all the threads in the grid as a result of launching your matrix multiply kernel? Please give the answers for reads and writes separately.
3. Assume that a kernel is launched with 1024 thread blocks each of which with 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel? (1 point) (A)1 (B)256 (C)512 (D)1024
4. For our tiled matrix-matrix multiplication kernel, if we use a 8x8 tile, what is the reduction of memory bandwidth usage for input matrices A and B? (A)1/8 of the original usage (B)1/16 of the original usage (C)1/32 of the original usage (D)1/64 of the original usage.
5. Suppose that the device on which your program runs has a hardware limitation such that each streaming multiprocessor (SM) can only accommodate up to 1536 threads and 16384 registers. Suppose your kernel uses a total of 12 local scalar variables. Suppose that you use a block size of 256 for your kernel. How many blocks can you fit in a SM at a time? How many threads can you fit in a SM at a time? Justify your answer and/or show your work to get full credit.
6. Suppose you want to use a tile size of 32-by-32 for your tiled matrix multiplication kernel, as implemented in lecture and the course text. Assume that your device has hardware limitations such that there are only 16,384 bytes of shared memory in an SM and the maximum number of blocks allowed per SM is 8. The maximum number of threads per SM is 3072. What is the maximum number of blocks that can be assigned per SM for your kernel? How about the maximum number of threads per SM? Justify your answer and/or show your work to get full credit. (Assume that each element in your matrices is a single-precision floating-point number.) Hint: Don’t forget that the shared memory in your kernel contains elements from both matrices A and B"

**Step 6:** Submit your assignment. You need only submit to Canvas the following files:

* answers.txt

Feel free to submit your source code along with answers.txt in a zip file, if you'd like, but only answers.txt is required. Please put the URL of your repo at the top of answers.txt, along with your name.

**3. Grading:**

Your submission will be graded based on the following criteria.

* Functionality/knowledge: 90%
  + Correct code and output results
  + Correct usage of CUDA library calls and C extensions
  + Correct handling of boundary cases
* Answers to questions: 10%
  + Correct answer to questions in step 5
  + Sufficient work is shown
  + Neatness and clarity