# CS 4444/6444 Spring 2017
# Assignment 5: Matrix-Matrix Multiplication with CUDA

In this assignment you will write an efficient CUDA implementation for the classic matrix-matrix multiplication problem. It is one of the most important problems, if not the most important problem, in high performance scientific computing and extensively studied in many different hardware architectures – including NVIDIA general purpose graphical processing units (or GPGPUs).

CUDA is NVIDIA's programming tool for programming on GPGPUs. It is mostly C with some additional features and annotations that only make sense to the GPU hardware. For a better understanding of the tool you can check the programming guide in NVIDIA's website at the following address.

***http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html***

Also refer to class materials on CUDA and GPUs if you need further help.

We expect all of you know the matrix-matrix multiplication problem. Nonetheless, to not leave it to chance, in a matrix-matrix multiplication problem, you compute entries of a result matrix C by multiplying two argument matrices A and B using the following rule.

$C_{ij} = \sum (A_{ik} \times B_{kj})$ for all $k$ in columns of $A$, $i$ in rows of $A$, and $j$ in columns of $B$

Your program should accept four command line arguments dictating the dimension lengths of the two argument matrices. The matrices will be randomly initialized. To ensure that your CUDA implementation produces the correct result, you must compare its output with the output of a CPU implementation. We are providing you with an inefficient CPU implementation in the *mmm_template.cu* file. In fact, you can use that file as your starting point and add your changes into it.

For the performance measurement, this time you will vary the input sizes – unlike in the HALO problem where you kept it fixed. You will use square argument matrices and calculate the average GPU runtime for 1000-by-1000 up to 10000-by-10000 matrices each time increasing the dimension lengths by 1000.

The second part of the performance measurement is to examine how performance varies as you change the kernel launch configurations by changing the number of grid-blocks and threads-per-grid-block parameters. For this part, you should use the 10000-by-10000 matrix instance. You must plot both measurements in graphs and reason about the behavior you observed.

Remember that the provided CPU implementation in the template file is inefficient. So do not get too excited if your first GPU implementation performs 100 times better than this, in particular, for larger matrices. In fact, I will not be surprise if someone writes a single-threaded CPU implementation with proper cache blocking that performs 100 times better than the one given. Further, you should verify the output of your GPU implementation with the given CPU implementation only for small matrix sizes, for 3000-by-3000 matrices for example. This is because the running time of the CPU implementation will

otherwise go quickly out of control and you have to wait almost an hour to get your performance result for a run.

Note that efficient memory access, cache blocking, and proper thread ordering are of paramount importance in CUDA programming. To give you an idea of how much performance varies between different GPU implementations of this problem for the same level of parallelism, my first GPU implementation on a GPU in the CS cluster for 7000-by-7000 square matrix was 70 times slower than my best implementation.

Unlike the previous heated plate problem, matrix-matrix multiplication problem, due to its popularity, provides many free solutions in the internet. Strictly avoid the temptation of searching for such solutions. As usual, do a lot of commenting on your implementation justifying your design decisions.

Not to leave you in the dark about CUDA programming, we are providing a CUDA solution of a different problem: vector addition. It is in the *vector_add.cu* file. Compile and run the program for different input sizes. It takes one command line argument for the length of the two vectors it adds together.

## Compiling and Running CUDA Programs

You need the ***nvcc*** CUDA compiler to compile any CUDA program. Note: use -O3. In the CS cluster, the power nodes have the nvcc compiler installed. To run your program, you need a machine having a GPU. In the CS cluster, the same three machines mentioned earlier each has a GPU attached to it. They have an older GPU card, but it is okay if you do your performance measurements there. The Artemis nodes have K20 GPUs. To access them include

```
#SBATCH --gres=gpu
```

to your SLURM/bash script.

Remember that understanding the architecture of your target platform is essential for good parallel programming. It is no different in CUDA programming. To find out the model number of the GPU in your target machine, use the ***nvidia-smi*** command. Then check the hardware configuration using that model number in the web. You already know how to use a SLURM script to find information about hardware features.

## Associated Files

1. **vector_add.cu**: a sample program to demonstrate how to do CUDA programming
2. **mmm_template.cu**: a template file having a CPU implementation of matrix-matrix multiplication and some handy supporting functions for data initialization and transfer