EE-193: Parallel Computing
Lab 3: Matrix multiplication

In this lab, you will experiment with multiplying matrices. The file matrix_mpy.cxx already contains:

- A class *Matrix*. This provides various basic capabilities:
  - Instantiate a matrix. You supply the "bits per dimension." E.g., *Matrix*(3) instantiates an 8x8 matrix and *Matrix*(5) instantiates a 32x32 matrix.
  - Various initialization routines. *Matrix*::*init_identity*() initializes it to the identity matrix; ::*init_random*() fills it with random numbers in [0,1]; and ::*init_cyclic_orde*r() fills the first row with [0,1,2,…], the second row with [1,2,3,…], the third row with [3,4,5,…], etc.
  - *Matrix*::*mpy_dumb*() implements a very simple matrix multiplication scheme that is single threaded and not blocked; it is simple and reliable but slow.
  - *Matrix::mpy1*() and *::mpy2*() are the functions for you to write.
  - *operator*(). Internally, the matrix stores its data as one long 1D vector, rather than as a 2D array. However, we give you *operator*() to access it easily with two indices. E.g., you can do *foo*=*my_matrix*(3,5), or *my_matrix*(3,5)=2. I had to use *operator*() rather than *operator*[], since the latter is only allowed to accept one parameter.
  - *row_str*(int *row*) returns a printable string for the given row; *str*() returns a printable string for the entire matrix.
- The function *main*() that calls matrix multiplication for you, times the work and checks the results.

You should implement:

- mpy1: a single-threaded blocked matrix multiply that uses the ordering rB, kB, cB, r, k, c.
- mpy2: a multithreaded implementation that spawns as many threads as requested.

The program matrix_mpy.cxx will then collect the following timing data:

- The *mpy_dumb*() algorithm for matrices of size 1Kx1K,
- *mpy_dumb*() and both of your algorithms for a matrix of size 2Kx2K. It will use a block size of 128x128; and run *mpy2*() using 1, 2, 4, 8 and 16 threads.

Provide a short report explaining your results as much as possible. You should at least discuss the following questions:

- *Mpy_dumb*() probably got almost 40x slower when moving from 1Kx1K matrices to 2Kx2K matrices. How did the number of floating-point operation change? Why might the time have increased so much more than the number of FLOPs?
- How much faster was your mpy1() algorithm than *mpy_dumb*()? How might you explain that?
- How did the addition of more cores affect your multi-threaded algorithm? Explain why.
- It runs out that a non-blocked algorithm just like *mpy_dumb*(), but with loop order *r*, *k*, *c* rather than *r*, *c*, *k* runs almost as well as *mpy1*(), even for fairly large matrices. Could you conjecture how this might possibly be? If we rewrote all our algorithms to use AVX instructions (Intel's SIMD instruction set introduced in 2011), we could get up to almost

8x the number of floating operations per core. If we did that, do you think that the non-blocked *r*, *k*, *c* algorithm would still run as fast as *mpy1*()?

Remember that each lab machine has one chip with four dual-threaded cores. Each core has its own 32KB L1D and 256KB L2, and there is an additional 8MB of L3 ring cache shared among the four cores.

***Logistics*:**

- The lab assignment is due ????? at midnight. You must work on this assignment on your own.
- Remember to compile with the line **c++ -pthread -std=c++11 -O2 matrix_mpy.cxx ee193_utils.cxx**. You may add debugging flags as needed.
- Submit your matrix_mpy.cxx via **provide**. You should also submit a copy of your report as a PDF.
- As usual, use any Linux box in labs 116 or 118. Before you collect benchmarking data, you should reboot the machine.