This document describes the structure of the matrix multiply project for 6.975. Because there are so few participants, we are going to keep the project rather open-ended, meaning that we will not provide you with any project code (a harness).

Your core implementation will multiply an $N \times 8N$ matrix by an $8N \times N$ matrix to produce an $N \times N$ matrix. On Tilera, the elements will be 32-bit integers. On Cell, the elements will be 32-bit (single-precision) floating point numbers.

The input matrices will begin in off-chip memory; use a statically initialized array or read the array elements from a file to an array in off-chip memory. Next, copy (i.e., stream, load, DMA) the entire input to on-chip memory (Cell’s SPU local stores; Tilera’s Cache Engines’ L2s). You have the freedom to choose whatever data distribution complements your decomposition. However, you are not allowed to duplicate any of the input data, meaning that only one copy of an element can exist on-chip. (Once the parallel matrix multiply begins, and timing begins, duplication is permitted.)

Next, your code will perform the parallel matrix multiply. There are no restrictions for this step. The performance characteristics of this step should be isolated and presented (see below).

Finally, the implementation should aggregate the output matrix in off-chip memory and check its correctness. For example, you might stream the output matrix back onto a single tile and check it against a sequential implementation.

The steps described above are for a single problem. You need to determine the largest problem size that your decomposition can support given that all input data must fit into on-chip memory. For Cell, this means the code and data must fit in the combined local store of the SPUs. If it does not, the program will generate an error. For Tilera, this means the code and data must fit in the combined L2 cache memory of the tiles. Since each of Tilera’s L2 caches is backed automatically by the main memory, you will have to determine the maximum problem size empirically, employing Tilera’s performance analysis tools (the MDE, Tile Processor Trace Viewer) to determine if you have frequent L2 capacity misses during the timed portion of your execution. If so, reduce the maximum problem size, re-run, and re-analyze until you have no L2 capacity misses during the timed portion of your execution.

You are required to run problems $1/64, 1/32, 1/16, 1/8, 1/4, 1/2$, and 1 of your maximum problem size. You are not provided with sample input matrices. Please create a system (harness) that will automatically run all problem sizes on input matrices with values of your choice. Your harness could be a Makefile, Perl program, or shell script.
Initial Report
The initial report is a one page document describing your progress and planned implementation. Briefly describe your proposed decomposition, mapping and maximum problem size search. You should also discuss any architectural benchmarks you created to test the capabilities of the target. For Tilera, the document might also cover which communication and synchronization mechanism(s) you plan to employ.

Structure of Your Implementation
For a single problem your implementation will perform the following steps:

1. Initialize values of input matrices. The matrices are resident in off-chip memory.
2. Copy the input matrices to on-chip memory.
4. Perform matrix multiplication.
5. End timing.
6. Display problem, timing, and performance information to user (see next section).
7. Aggregate output matrix in off-chip memory.
8. Check correctness of output matrix and report to user.

This core implementation will be wrapped in a harness that will run it for the multiple problem sizes discussed above.

Performance Analysis
For each problem size you are required to gather performance statistics for the matrix multiply portion of your implementation. These statistics must include:

- Runtime in cycles
- Total FLOPs/OPs,
- Compute utilization (SPU instructions per cycle; Processor Engine instructions per cycle)
- Instruction breakdown (computation, communication, overhead, blocked)

The runtime is defined as the maximum number of cycles across all cores. You might want to use a global barrier immediately before you end your cycle counter. Since we are not providing you with a harness, you are responsible for inserting all timing/performance instrumentation. For Tilera, you might want to use the Profiler Control API Reference described in the documentation included in the distribution (/opt/TileraMDE-1.1.1/doc). For Cell, you can use the decremenenter counter of each SPU.

Feel free to add any other quantities. For the untimed portion you must also report how many cycles it took to copy the input data to on-chip memory and copy the output data to off-chip memory. This requirement is not explicit in the structure given above.

These statistics will be presented in your final document (see below). You do not have to generate all of the statistics during a terminal execution of your project; many will require the use of a performance
monitoring tool. However, please display whatever you can gather at runtime to the user such as total runtime.

**Final Document**

The final document will describes your decomposition, implementation and results in at least five pages (single-spaced, 2 columns). Do not neglect to include references to any materials used for the project. The document must include sections covering:

- A description of the architecture and programming system
- The architectural benchmarks performed and how they guided your parallelization strategy
- Your decomposition, mapping, and data distribution
- The implementation including code structure and communication mechanisms
- What worked and what did not (lessons learned)
- Breakdown of the impact of different optimizations/versions of your implementation
- Performance analysis for each problem size and summary charts
- Qualitative analysis of your effort

**What to hand in**

1. Your complete, **well-documented** source code including a Makefile system to build your source code into an executable and a harness to run the multiple problem sizes
2. Sample input matrices for each problem size that will be used by your harness
3. Final document

**Submission Procedure**

Please email your Initial Report to 6975-staff@lists.csail.mit.edu. For the submission of the final project, please archive your code and your Final Document and email it to the same address. Alternatively, you can provide us with a url from which to download the project. Make sure the Final Document and all code documentation is easy to find in the package.

**Competition**

For each architecture, we will present an award to the student whose in-core matrix multiplication implementation is the fastest. The competition will use problem sizes determined by the 6.975 course staff, normalizing across student implementations. We will not reveal anything further about the competition until after the winners are determined.

**Bugs**

Please report all bugs in either tool chain to the TA, Michael Gordon, mgordon@mit.edu. He will then forward the bug report to the responsible party.