P-Last Matrix Multiply

Regardless of your selection, all versions should utilize shared memory, coalescing memory, and with the assumption that matrixes will be a multiple of the block size with associated tile-size, we do not have to deal with divergence. The input and output file specs are the same as in P1 and P2. Please make sure the output result matrix has the N, the matrix size, on the first output line and each of the remaining N lines contain N floats using a field of 6 characters wide with 2 digits to the right of the decimal point, %6.2f. (In C++, setprecision(2), setw(6) in fixed notation). Each line ends with a newline (including the last line). There should be no additional spaces anywhere.

Programs will be tested with 6 test files and 5 tile sizes

- with a fixed tile size of 32x32 and matrixes as follows 32x32, 256x256. 512X512, 1024x1024, 2048x2048, 4096x4096
- with a fixed matrix of size 4096x4096 tested with the following tile sizes 4x4, 8x8, 16x16, 32x32, and 64x64.

It should fail gracefully with a tile size of 64x64 and should be caught by your code.

The last lecture, the class decided to report with two graphs to compare the results of P1, P2, and P-last. Both graphs will utilize a 4096x4096 matrix with various tile sizes of 4x4, 8x8, 16x16, and 32x32.
1) Graph 1 – compare the overall times – for the 4 tilings in P1, P2, and P-last.
2) Graph 2 – compare the times of Cuda – (the Cuda setup and kernel call) – for the 4 tilings in P1, P2, and P-last. (make sure you only include cuda setup, kernel call, and tear down in this time).

It was suggested to use time on the y axis and different tilings on the x-axis. For each tiling starting at 4x4, have 3 bars, one for each kernel. The idea here is to observe if there is any difference between the overall kernels and within a particular kernel, does a tile size make a difference.

NOTE: If P1 and P2 are working, you should be able to implement just a new kernel.

Here are the options on the kernel. I am providing an outline which may or may not need to be tweaked. The idea is correct but you will have to read between the lines.

B-level~~ – two options:~~
Option 1: Cuda technology focusing on double buffering:
Perform double buffering while the loading of values into the matrix. The idea is to use a register to buffer the data flowing into shared memory. From global memory to a register and then later from a register into shared memory, i.e. double buffering.

```
setup shared memory
prefetch the first two values into registers (local variables), i.e. m=0
loop over the (width/tile_width -1) tiles staring at m=1{
    move current two values from registers into shared memory
    sync
    prefetch the next two values into registers

    loop - process the tile that's in shared memory
    sync – make sure all threads have finished
}
move last values from registers into shared memory
sync
loop - process the tile that's in shared memory
sync – make sure all threads have finished
Store the final value.
```

~~Option 2:~~ UPDATED TO A level: Cuda technology focusing on granularity – loop unrolling:
Make each thread do more work. Use one thread to compute 4 elements of the solution matrix. The basic idea is to make the code more granular by performing more work in each thread. The general flow of the kernel is

```
setup shared memory
loop over the tiles {
    each thread loads
        4 values of a tile of matrix M
        4 values of a tile of matrix N
        into shared memory
    sync – make sure all 8 values are loaded for each thread.
    process the tile – 4 values in the solution matrix are updated
    sync – make sure all threads have added to their respective 4 values
}
store the 4 values in the solution matrix
```

~~A -level~~ Update to A+ level: with cuda technology focusing on granularity, double buffering and data movement:
P last is a modified version of the last two versions where we incorporate the idea of pre-fetch (or double buffering) and alter the order we multiply the values. The idea is to intersperse the data movement with the multiplication process. The idea was presented in a previous class. While performing the mult of Md * Nd = Pd, we will mult and fetch based on

```
|A B|  |1 2|  |W X|
|C D|  |3 4|  |Y Z|

Load into shared mem
A, 1, 2
C, 1, 2
B, 3, 4
D, 3, 4
```

Here is the basic idea - don't hold me to the outline and remember the syncs are not in place. You have to carefully place the syncs yourself.

W=X=Y=Z=0

```
pre-fetch values from each  A, 1, 2
loop through the blocks as we have done most of the semester
    move the pre-fetched values from A, 1, 2 into shared mem
    pre-fetch a value for C
    loop do
        W += A*1
        X += A*2
    move the pre-fetched value from C into shared mem
    pre-fetch values from each B, 3, 4,
    loop do
        Y += C*1
```

```
            Z += C*2
        move the pre-fetched values from B, 3, 4 into shared mem
        pre-fetch D
        loop do
            W += B*3
            X += B*4
        move the pre-fetched value from D into shared mem
        pre-fetch next values from each A, 1, 2,
        loop do
            Y += D*3
            Z += D*4
    end of loop
    store 4 values into global mem
```

What to submit?
1) The source code containing the kernel/device code. I will compile and supply my own data sets to test your kernel for correctness. FOLLOW the output format exactly.

2) Report the results as noted above.