LSU EE 7722 **Homework 2** **Due: 8 March 2017**

Problem 0: Read the following information about the assignment package, and follow instructions on course procedures page, `http://www.ece.lsu.edu/gp//proc.html`, for account setup and Programming Homework Workflow. Try compiling and running the code and familiarize yourself with the command line arguments described below.

The code in `hw02.cu` computes matrix squares. The input is an array of S (default $S = 16384$) $N \times N$ matrices (default 16×16) of floats. Let A denote element i, $0 \leq i < S$, of the input. Element i of the output is set to the matrix product A^2. The kernels in `hw02.cu` compute a matrix product using the usual three-level loop nest.

When the program is run without arguments (typing `./hw02` at the command line) it will launch each kernel multiple times with different block sizes. Arguments can be given to control the number of blocks, the block size, and the data size. The program output starts with data about the GPUs that it will use:

```
GPU 0: GeForce GTX 1080 @ 1.73 GHz WITH 8113 MiB GLOBAL MEM
GPU 0: L2: 2048 kiB   MEM<->L2: 320.3 GB/s
GPU 0: CC: 6.1  MP: 20  CC/MP: 128  DP/MP:  4  TH/BL: 1024
GPU 0: SHARED: 49152 B/BL  98304 B/MP  CONST: 65536 B  # REGS: 65536
GPU 0: PEAK: 4438 SP GFLOPS  139 DP GFLOPS  COMP/COMM:  55.4 SP  3.5 DP
GPU 1: Tesla K20c @ 0.71 GHz WITH 5060 MiB GLOBAL MEM
GPU 1: L2: 1280 kiB   MEM<->L2: 208.0 GB/s
GPU 1: CC: 3.5  MP: 13  CC/MP: 192  DP/MP: 64  TH/BL: 1024
GPU 1: SHARED: 49152 B/BL  49152 B/MP  CONST: 65536 B  # REGS: 65536
GPU 1: PEAK: 1761 SP GFLOPS  587 DP GFLOPS  COMP/COMM:  33.9 SP  22.6 DP
Using GPU 1
```

Most lab computers have two GPUs, please pay attention to the GPU that is actually being used, GPU 1 above.

The execution rates shown above (GFLOPS) count a multiply-add as one operation. The `COMP/COMM` line gives the computation to communication ratio in floating-point operations per floating-point element transfers. The assignment code uses SP (single precision floating point) by default. (The information above was collected in part using the runtime library's `cudaGetDeviceProperties` function.)

The program will next print information about each kernel:

```
CUDA Kernel Resource Usage:
For mxm_g_only:
       0 shared, 48 const, 0 loc, 38 regs; 1024 max threads per block.
For mxm_g_split<1>:
       0 shared, 48 const, 0 loc, 38 regs; 1024 max threads per block.
For mxm_g_split<8>:
       0 shared, 48 const, 0 loc, 38 regs; 1024 max threads per block.
```

Next the program prints information about the input size and the launch configuration (number of blocks [grid size] and block size):

```
Input is 16384  16 x 16 matrices of float, total size 16777216 bytes (16.0 MiB).
Launching with 13 blocks of up to 1024 threads.
```

The matrix size (16 above) can be changed by editing the assignment to `N` near the top of the file. For performance reasons it's important that it be a compile-time constant (as opposed to something set based upon input or varied in a loop).

The code takes three command-line arguments. The first indicates how many blocks to launch. If the argument is zero then the number of blocks will be set to the number of multiprocessors (which is the default). If the argument is negative then the number of blocks will be $-aM$, where a is the argument value and M is the number of MPs.

The second argument is the number of threads per block to try to use to launch each kernel. If the argument is omitted 1024 threads are tried. If the argument is omitted or positive, the actual number of threads used in a launch is the minimum of this argument and the kernel's maximum. (For example, if the second argument is 512, but kernel `foo` has a limit of 256 threads, `foo` will be launched with 256 threads.) If the second argument is zero then each kernel will be launched multiple times starting with 4 warps, incrementing by 4 warps until the kernel maximum is reached.

The third argument indicates the size of the input in units of MiB. The default is 16 MiB. If a_3 is the value of the third argument, the input size will be $a_3 2^{20}$ B. The third argument is read as a floating-point number, so "0.5" will result in a 2^{19} B input.

Here are some examples: Run with 256 threads per block: `./hw02 0 256`. Run with 512 threads per block and twice as many blocks as MPs: `./hw02 -2 512`. Run with 256 threads per block and 10 blocks: `./hw02 10 256`. Run each kernel multiple times using an input size of 1 GiB: `./hw02 0 0 1024`.

Problem 1: The routine in `mxm_g_only` computes A^2 inefficiently, especially for those devices without a read-only cache.

(a) Find an expression for the ratio of the actual amount of data crossing the GPU chip boundary to the ideal amount of data crossing the chip boundary (see below) ignoring caches. Show the ratio in terms of the following symbols:

N Matrix dimension. (The matrix has N rows and N columns.)
S Number of matrices.
E Element size. (Four for float, eight for double).
R Minimum request size.
B Block size (number of threads per block).
G Grid size (number of blocks).
M Number of MPs.

The ideal amount of data is simply the size of the input and output, SN^2E bytes read and SN^2E bytes written, for a total of $2SN^2E$. For this problem base the actual amount of data crossing the chip boundary on the minimum request size and the number of times an element is read (ideally, once).

A ratio of 2 indicates that the system is reading twice as much data as needed. A ratio of 1 is ideal, and any ratio below 1 is impossible since it would indicate either that not all data had been read or not all data had been written.

Hint: The expression should be fairly simple. Things get interesting in the next problem.

(b) Run the code and compare the execution time of `mxm_g_only` to your expectations based on the analysis above. Your expectations should be based on the chip data bandwidth (shown as `MEM<->L2`) and other relevant factors. Please indicate the type of GPU on which the code is run. (Note: See the last part of the next problem.)

Problem 2: In routine `mxm_g_only` each A^2 computation is performed by one thread. (But each thread does multiple A^2 computations.) In this problem `mxm_g_split` will be modified so that each

computation is performed by `thds_per_mat` threads, where `thds_per_mat` is a template variable. In some situations the reason for having multiple threads perform a unit of work (such as computing A^2) is to accommodate a smaller amount of work (fewer matrices) while still having enough threads to keep the hardware busy. But in this problem there is another benefit.

The `main` routine runs `mxm_g_only` and two specializations of `mxm_g_split`: for `thds_per_mat` set to 1 and for `thds_per_mat` set to 8. In the unmodified file the contents of `mxm_g_split` is identical to `mxm_g_only` and so all three run times should be very close.

Remember that a template variable appears as a constant to the compiler (unlike an argument). That's important here since it enables the compiler to most effectively apply loop unrolling and other techniques when the loops are a function of `thds_per_mat`.

(*a*) Modify `mxm_g_split` so that `thds_per_mat` threads together perform a single A^2. The code must work correctly when `thds_per_mat` is set to a power of $2 \leq N$. Do so by changing the loops, but don't try to use shared or local memory. For example, suppose `thds_per_mat` is set to 2. Then, say, the even threads might compute rows 0 to $N/2 - 1$ and the odd threads rows $N/2$ to $N - 1$. However, that's not the best way of doing things. Instead split work between the threads to minimize the amount of data read.

(*b*) Based on code for the part above, find an expression for the ratio of the actual amount of data crossing the chip boundary to the ideal amount of data. Use the symbols from the last problem but also use symbols d for the value of `thds_per_mat`.

(*c*) Based on your analysis in the previous part find the smallest value of `thds_per_mat` that minimizes the ratio. Consider the minimum request size and the warp size.

(*d*) Check whether your minimum `thds_per_mat` predicts actual performance running specializations of `mxm_g_split` at sizes less than, equal to, and larger than your ideal. If necessary, increase `N` (near the top of the file). Add specializations by modifying code near the word `specializations`.

(*e*) Your analysis above should have ignored all sorts of caches. In reality, all GPUs have L2 caches and some GPUs have L1 caches. Indicate what caches were available in the GPU you ran your experiments on. (Note: the code is written so that a read-only cache can be used for input data.)

Indicate whether performance results suggest that caches were doing any good. Information on the available caches can be found in the CUDA Programmer's guide, especially in Appendix G, hardware implementation. Note that the CC (compute capability) of the available GPUs are printed when the program starts.