LSU EE 7722 **Homework 3** **Due: 29 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 Program

The code in `hw03.cu` computes matrix products. The input is two arrays of S (default $S = 16384$) $N \times N$ matrices (default 8×8) of floats. Let A and B denote element i, $0 \leq i < S$, from each of the inputs. Element i of the output is set to the matrix product AB. The kernels in `hw03.cu` compute a matrix product using a method based on an algorithm in Volkov 2008.

When the program is run without arguments (typing `./hw03` 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:
CUDA Kernel Resource Usage:
For mxm_volk:
   32768 shared, 64 const, 0 loc, 42 regs; 1024 max threads per block.
For mxm_tpc<1>:
   32768 shared, 64 const, 0 loc, 42 regs; 1024 max threads per block.
For mxm_tpc<2>:
   32768 shared, 64 const, 0 loc, 42 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 32768 pairs of 8 x 8 matrices of float,
```

```
  total size 16777216 bytes (16.0 MiB).
Launching 13 blocks of 1024 threads with 19.69 matrices per thread.
```

The matrix size (8x8 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`.

When run without arguments the program runs each kernel with a varying number of works producing the following output:

```
Kernel mxm_volk:
 4  4 wp     564 s    30 GF    45 GB/s ===========
 4  4 wp     564 s    30 GF    45 GB/s  2.69 I/F   0.3 wp/c    0%   1.6
 8  8 wp     317 s    53 GF    79 GB/s ===================
 8  8 wp     317 s    53 GF    79 GB/s  2.69 I/F   0.8 wp/c    0%   1.6
12 12 wp     253 s    66 GF   100 GB/s ========================
12 12 wp     253 s    66 GF   100 GB/s  2.69 I/F   1.1 wp/c    0%   1.6
```

For each kernel execution two lines are shown, which are identical up to the ASCII art bar graph. The first column shows the number of warps per block, the second column shows the number of warps per MP. The number of warps per MP will be higher when the kernel is launched with more blocks than MPs and when there are sufficient resources for more than one block. For this problem the two resources limiting block occupancy is the amount of shared memory used and the number of registers used.

The numbers to the right of `wp` show: execution time, FP throughput in units of billions (10^9) multiply/add operations per second, and data throughput in GB/s. For the meaning of the other numbers inspect the code and see which metrics are being collected.

Matrix/Matrix Multiplication

Matrix-matrix multiplication is an important problem to consider when studying computational accelerators, including GPUs, because it is simple and so amenable to hand analysis, the nature of the problem changes with the dimension of the matrices, and because it is a major component of many important scientific computations.

Multiplying two $N \times N$ matrices in the straightforward manner requires N^3 multiplies and $N^3 - N^2$ additions. In this analysis we will count that as N^3 multiply/add operations, in part because in the computational accelerators of interest, NVIDIA GPUs and the Intel Xeon Phi, a

fused multiply/add instruction has the same throughput and latency as a multiply instruction or an add instruction. That is, the best possible execution time for N^3 multiply/add instructions is the same as that of N^2 multiply instructions plus $N^3 - N^2$ multiply/add instructions.

The amount of data transferred when multiplying two $N \times N$ elements is $3N^2$ elements.

As we've discussed in class common performance measures for computational devices are data bandwidth (48.1 GB/s), floating-point bandwidth (772 SP GFLOP/s or 16 DP GFLOP/s), and instruction bandwidth (384×10^9 insn/s), all sample performance numbers are for my laptop equipped with a Quadro K2100M.

For both computational devices and computational problems we can compute a *computational intensity* also known as a *computation to communication ratio*. For a device is (usually) the FP bandwidth divided by the data bandwidth, with the bandwidth units usually chosen for single or double precision computation. For example, for the K2100M, the ratio for single-precision computation is $\frac{772\,\text{SP GFLOP/s}}{\frac{1}{4}41.1\,\text{GB/s}} = 64$, and for double-precision computation is just $\frac{32\,\text{DP GFLOP/s}}{\frac{1}{8}48.1\,\text{GB/s}} = 5.32$.

For the $N \times N$ matrix multiplication problem the computational intensity for the single-precision variant is $2N^3/(3N^2) = 2N/3$ floating-point operations per element.

By equating the computational intensity expressions for the device and the matrix multiplication problem we can find the dividing line between matrix sizes that are data-limited and those that are computation limited: $\frac{\theta_{\text{sp}}}{\frac{1}{4}\theta_{\text{data}}} = \frac{2N}{3}$, yielding $N = \frac{6\theta_{\text{sp}}}{\theta_{\text{data}}}$, where $\theta_s p$; is the single-precision bandwidth and θ_{data} is the off-chip data bandwidth. For the the K2100M $N = 96.3$ for SP and $N = 7.98$ for DP.

For the K2100M, when $N < 96.3$ the problem is data-limited and care must be taken to minimize redundant loads (or stores) of elements. When $N > 96.3$ the problem is FP-limited and care must be taken to reduce the number of instructions other than the multiply/adds.

Routine `mxm_volk` is a matrix-matrix multiplication routine based presented in Volkov 08 Supercomputing, a seminal paper on performing dense linear algebra on NVIDIA GPUs.

Routine `mxm_tpc` is initially the same as `mxm_volk` but will be modified in the second problem.

The makefile generates three executables: hw03, hw03-debug, and hw03-cuda-debug. The first is compiled with optimization on for host and GPU code. Executable hw03-debug is compiled with optimization off for CPU code and on for GPU code. Use it for debugging CPU code. Executable hw03-cuda-debug has optimization off for CPU and GPU code. Use it to debug GPU code. Use `gdb` for debugging CPU code and `cuda-gdb` for debugging both CPU and GPU code. Performance will be slower using `cuda-gdb` so don't use it for collecting performance data.

The makefile generates PTX (intermediate code) and SASS (assembly code) for the optimized version. The compiler *mangles* the name of the `mxm_tpc` kernels, and these will be used in the SASS output. Use command-line utility `c++filt` to *demangle* the name. For example, running `c++filt _Z7mxm_tpcILi1EEvPf` returns `void mxm_tpc<1>(float*)`.

Documentation and References

CUDA documentation can be found at `http://docs.nvidia.com/cuda/`. Details on the GPU instruction set can be found in the CUDA Binary Utilities manual. Information on the metric that can be collected by the NPerf calls see the CUPTI manual. Some information on the hardware can be found in the CUDA C Programming Guide Chapter 5 (especially 5.4) and in Appendix G. For detailed hardware descriptions follow the Accelerator Descriptions link on the course home page.

Problem 1: Characterize the performance of `mxm_volk` over a range of matrix sizes on a GPU of compute capability 3.0 or later. Be sure to include at least one matrix size that is computation-bound and at least one that is communication bound *for the device which you have chosen.*

When performing the runs pay attention to the number of matrices per thread and be sure that work is reasonably balanced. Use the third command-line argument if necessary to adjust the amount of work. Experiment with configurations in which there is more than one block per MP.

Show the performance of your best runs. Indicate how close performance was to the performance limiter. Based on the metrics and inspection of the SASS code indicate what you think prevented performance from being closer to the data or FP limit.

Pay attention to the following factors: Whether global loads used the read-only cache. (`LDG` instructions usually use the ROC, while `LD.E` do not.) Whether local memory was used. The distance between an instruction that writes a register and the instruction that uses that register value.

Problem 2: As noted in class, one weakness with `mxm_volk` is that each thread must overlap the computation of N elements of the output matrix. For larger values of N the compiler will run out of registers, forcing it to use local memory to implement the local-address-space array `elt`, rather than registers. As we know the latency of local memory loads is literally orders of magnitude larger than register access.

Modify routine `mxm_tpc` so that is uses c threads per column, where c is the value of template parameter `thd_p_col`.

Compare the performance to the original code. In Homework 4 your routine and the original will be analyzed in detail.