

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 homework package is set to compile for an NVIDIA GPU of compute capability 3.5 (the expensive Kepler). It is recommended that you run your code on such devices (including the machines in the lab). An easy way to determine the CC of the GPU in a lab machine is to consult the computer status Web page, <http://www.ece.lsu.edu/koppel/gpup/sys-status.html>. If you must run on a less capable machine edit the makefile, changing `sm_35` to the CC of your machine.

Each run of the code launches all of the kernels. A kernel may be launched once, or if the second argument is 0 (see below) launched multiple times with different block sizes. The program output starts with data about the GPUs that it will use:

```
Using GPU 0
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 MiB GLOBAL MEM
GPU 0: L2: 1310720 kiB MEM<->L2: 208.0 GB/s
GPU 0: CC: 3.5 MP: 13 CC/MP: 192 DP/MP: 64 TH/BL: 1024
GPU 0: SHARED: 49152 B CONST: 65536 B # REGS: 65536
GPU 0: PEAK: 1761 SP GFLOPS 587 DP GFLOPS COMP/COMM: 33.9 SP 22.6 DP
Using GPU 0
```

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 by default. Please don't try using DP in this assignment. 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 lane_aligned:
    0 shared, 64 const, 0 loc, 10 regs; 1024 max threads per block.
For scheduler_bm:
    0 shared, 64 const, 0 loc, 7 regs; 1024 max threads per block.
```

Next the program prints the array size and launch configuration:

Launching with 13 blocks of up to 1024 threads for 1048576 elements.

If the second argument was non-zero then each kernel is run once. The number of warps used to launch it is shown, along with execution time, and communication rate. The communication rate is based on an ideal amount of off-chip data transfer.

```
K lane_aligned      2 wp      688.288 μs    12.188 GB/s
K scheduler_bm     2 wp      780.672 μs    10.745 GB/s
```

If the second argument is zero then each kernel is run multiple times and an ASCII-art bar graph is printed.

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 number of input and output vectors in mibi-elements. If a_3 is the value of the third argument, the number of array elements will be $a_3 2^{20}$. The third argument is read as a floating-point number, so "0.5" will result in a 2^{19} vectors.

Here are some examples.

Running without arguments: `hw01`. This will use M blocks, where M is the number of multiprocessors, with up to 1024 threads per block. One could get the same result by running using `hw01 0 1024` or `hw01 M 1024` where M is replaced by whatever the number of multiprocessors is.

Run with 256 threads per block: `hw01 0 256`. Run with 256 threads per block and 10 blocks: `hw01 10 256`. Run each kernel multiple times: `hw01 0 0`.

Problem 1: In `mxv_g_only` the matrix is in constant memory. In this problem let's try device memory in `mxv_g_only_g`.

- (a) Modify `hw02.cu` so that kernel `mxv_g_only_g` reads the matrix from global memory rather than constant memory.
- (b) How does the performance differ for the default matrix size, 16×16 ?
- (c) Above some matrix size the performance of `mvx_g_only_g` should drop in comparison to `mvx_g_only`. Describe what the problem is.
- (d) Determine if in fact the expected change occurs. Inspect the SASS code to find changes in instructions and check performance above and below the threshold size. Comment on the number of instructions needed.

Problem 2: In `mxv_g_only` the matrix is in constant memory. In this problem let's try shared memory in `mxv_g_only_s`.

- (a) Modify `hw02.cu` so that kernel `mxv_g_only_s` initially reads the matrix from constant memory and places it in shared memory, then accesses it from shared memory when performing matrix/vector multiplications.
- (b) How does the performance differ for the default matrix size, 16×16 ?
- (c) Comment on the difference in the number of instructions and the impact that might have on the peak execution time.
- (d) Above some matrix size the performance of `mvx_g_only_s` should be much better than `mvx_g_only`. Explain why this might happen.
- (e) Determine if in fact the expected change occurs in `mvx_g_only`. Inspect the SASS code to find changes in instructions and check performance above and below the threshold size.

Problem 3: The kernels in this assignment make inefficient use of memory requests when loading and storing vectors.

The problem can be solved by interleaving the vectors. If the array is interleaved degree d then the first d elements of `d_in` and `d_out` will be the first component (say, v_x) of the first d vectors. Without interleaving, the program stores vectors contiguously in arrays `d_in` and `d_out`.

For example, the contents of `d_in` for $N = 2$ might start: $v_{0,x}, v_{0,y}, v_{1,x}, v_{1,y}, v_{2,x}, v_{2,y}, v_{3,x}, \dots$, where $v_{0,x}, v_{0,y}$ are the x and y components of the first vector.

If instead the arrays were interleaved degree 3 we would have: $v_{0,x}, v_{1,x}, v_{2,x}, v_{0,y}, v_{1,y}, v_{2,y}, v_{3,x}, \dots$ ■

(a) Modify the code so that `mvx_g_only_interleave` reads data with a degree-IF interleave factor. Note that `IF` is declared at the top of the file.

- Add new input and output arrays to `App` for use by the kernel.
- Modify `main` so that the interleaved array is based on `d_in`.
- Modify `main` so that the data read from the GPU for `mvx_g_only_interleaved` is put back in uninterleaved order, so that it can be checked for correctness.
- Modify `mvx_g_only_interleaved` so that it operates on the interleaved data.

(b) Comment on the performance of the routine. Determine how close you come to saturating FP and data transfer.