LSU EE 7722 **Homework 1** Solution **Due: 26 February 2016**

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 output below just shows two kernels:

```
Kernel lane_aligned:
 4 wp   0 acwp     352 μs     24 GB/s *****
 8 wp   0 acwp     184 μs     46 GB/s **********
```

```
12 wp   0 acwp     131 μs     64 GB/s ***************
16 wp   0 acwp     104 μs     80 GB/s ********************
20 wp   0 acwp      90 μs     94 GB/s ************************
24 wp   0 acwp      85 μs     98 GB/s *************************
28 wp   0 acwp      74 μs    114 GB/s ****************************
32 wp   0 acwp      72 μs    116 GB/s ****************************
Kernel scheduler_bm:
 4 wp   0 acwp     398 μs     21 GB/s *****
 8 wp   0 acwp     209 μs     40 GB/s **********
12 wp   0 acwp     147 μs     57 GB/s **************
16 wp   0 acwp     114 μs     73 GB/s ******************
20 wp   0 acwp      97 μs     87 GB/s *********************
24 wp   0 acwp      89 μs     94 GB/s ************************
28 wp   0 acwp      80 μs    104 GB/s ****************************
32 wp   0 acwp      75 μs    112 GB/s ****************************
```

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: Kernel `lane_aligned` reads an element from input array `d_app.d_in`, adds something to it, and writes the sum back to `d_app.d_out`. That something is $\frac{L}{10000}$, where L is the lane number of the thread. (The lane number of a thread is the `threadIdx.x` modulo 32, or equivalently, the last five bits of the binary representation of `threadIdx.x`.)

The code launching `lane_aligned` checks to make sure that the correct lane number is used. Furthermore, it expects that regardless of the block size, elements are written only by threads in a fully populated warp. A fully populated warp has 32 active threads. That means, for example, for a block size of 48, threads with a `threadIdx.x` value from 0 to 31 are in a fully populated warp, but threads 32 to 47 are in a partially populated warp and so should not execute. If they do execute an incorrect-result error will be reported.

One can see the error by running with the command `run 0 48`.

(*a*) Modify kernel `lane_aligned` so that only threads on fully populated warps execute while still continuing to operate on all elements of the array. A simple way of ending a thread's execution is

with a `return` statement. One can also use an `if` statement to guard the loop.

(*b*) Determine if there is any penalty in launching with partially populated warps in the solution to the previous problem. That is, is a launch with a block size of 260 run slower than a launch with a block size of 256? Show data from runs to justify your answer.

The table below shows the run time of 32- and 48-thread configurations. The first two are launched with one block per MP, the second with eight blocks per MP. Since the run times are very close there is at best only a tiny penalty for launching with unused threads.

```
K lane_aligned     1 wp     1403.584 s     5.977 GB/s  <- 32 thds, 1 bl/mp
K lane_aligned     2 wp     1399.552 s     5.994 GB/s  <- 48 thds, 1 bl/mp
K lane_aligned     1 wp      185.728 s    45.166 GB/s  <- 32 thds, 8 bl/mp
K lane_aligned     2 wp      182.752 s    45.902 GB/s  <- 48 thds, 8 bl/mp
```

Problem 2: The first column of the table produced when running with a zero as the second argument, say `hw01 0 0`, shows the number of warps in a block. The intent of the table is to show how the number of threads (or warps) impacts performance. But, the number of warps in a block is not the full picture. A MP (or an em pee) can have multiple *active* blocks, so the table can be misleading, for example, when there are 8 4-warp blocks per MP.

(*a*) Modify routine `main` so that the table shows the number of active warps per MP in the second column. (That column currently shows zeros.) To do this one must consider how many blocks are available per MP, M/G, where M is the number of MP's and G is the number of blocks (the grid size), as well as how many blocks will fit on an MP. For convenience the maximum number of blocks per MP has been written to variable `max_bl_per_mp`.

See the checked in code. Sample output appears below. Note that the number of active warps explains the irregular length of the bars.

```
Ran using: ./hw01sol -8 0

Kernel lane_aligned:
 4 wp  32 acwp      74 μs    114 GB/s ****************************
 8 wp  64 acwp      60 μs    140 GB/s ************************************
12 wp  60 acwp      62 μs    135 GB/s **********************************
16 wp  64 acwp      61 μs    138 GB/s ************************************
20 wp  60 acwp      62 μs    136 GB/s **********************************
24 wp  48 acwp      64 μs    131 GB/s ********************************
28 wp  56 acwp      62 μs    135 GB/s **********************************
32 wp  64 acwp      61 μs    138 GB/s ************************************
```

(*b*) Using the modified code determine the minimum number of warps needed to achieve close to maximum performance on kernel `lane_aligned`, and determine whether it helps or hurts to have more than one active block per multiprocessor. (Consider only block sizes that are a multiple of 32 threads.)

The data below is from runs in which the number of blocks per MP 2, 4, and 8. Each line shows the performance when there are 64 active warps. Looking only at the third group (104 blocks), it appears that having 8 blocks of 8 warps is faster than 2 blocks of 32 warps, so there is no penalty for having more blocks per MP. The data also show a slight advantage for configurations with fewer threads. That might be due to secondary factors.

```
Launching with 26 blocks of up to 1024 threads for 1048576 elements.
32 wp  64 acwp      59 μs    142 GB/s ************************************
```

```
Launching with 52 blocks of up to 1024 threads for 1048576 elements.
16 wp  64 acwp      59 μs    141 GB/s ***********************************
32 wp  64 acwp      60 μs    139 GB/s ***********************************
Launching with 104 blocks of up to 1024 threads for 1048576 elements.
 8 wp  64 acwp      60 μs    140 GB/s ***********************************
16 wp  64 acwp      61 μs    139 GB/s ***********************************
32 wp  64 acwp      61 μs    138 GB/s ***********************************
```

Problem 3: Kernel `scheduler_bm` includes an iterative calculation over a sine operation. In particular, it uses an intrinsic for NVIDIA's approximate sine machine instruction. (Actually a pair of instructions.)

According to the NVIDIA C Programming Guide (v 7.5, Section 5.4) a CC 3.X device can issue 32 sine operations per cycle per MP, which works out to 8 per scheduler.

A warp with 32 active threads should take 4 cycles to issue a sine instruction. In this problem determine whether a warp with fewer active threads takes less time. For example, would a warp with only eight active threads take just one cycle to issue? If so, does it matter which lanes those active threads are in?

(*a*) First things first. Running with `hw01 0 0` one might notice that both kernels take the same amount of time despite the fact that `lane_aligned` just does one add, while `scheduler_bm` does two sine operations. Even if the approximate since operation takes the same amount of time as a FP add, the results still don't make sense because *two* sine operations are being performed. Explain why the execution times are similar and modify the code so that the impact of the sine operation is exposed.

Try to work things out with the following assumptions. Global memory access latency is 200 cycles. FP add latency (and for that mater, multiply and multiply/add) is 11 cycles, and the sine operation is 40 cycles. Also assume that the GPU can transfer at most 300 bytes per cycle on and off chip. (That's about 200 GB/s at .71 GHz.)

Two factors are hiding the impact of the sine operation: latency and memory traffic.

Consider the first `i` iteration of thread 0. Since it's at the front of the line for everything, the time it takes to finish that first iteration is based only on latency. However, over that time all other threads in active warps have made requests to load stuff from memory. Thread 0 can't really start its second iteration until all of that loaded data arrives. It's worse when thread 0 starts its third iteration, because it needs to wait for the memory system to finish with loaded and stored data over the previous iteration.

Show how the impact of data bandwidth can be removed (a convenient knob has been provided) and show how the impact of sine issue rate can be exposed.

Briefly: increase the value of `iters` so that execution time is dominated by the sine. There are 32 special units, which works out to 8 per scheduler on a Kepler device. So each scheduler requires four cycles per warp. If the sine latency is 40 cycles, then we need ten warps per scheduler or 40 per MP to saturate issue, and more than that to see the impact of reduced issue time. That's not problem because we can easily run with 64 warps per MP.

(*b*) Using the version of the code with the impact of the sine issue rate exposed, determine if warps issuing a sine instruction take fewer than 4 cycles to issue if they have fewer than 32 active threads.

A mask value can be used to select which threads should be inactive (or active). For example, if `threadIdx.x & mask` is non-zero one might choose to make the thread inactive. By inactive, we only mean that it does not perform the sine operations. It is important to do this in such a way that an inactive thread reads and writes memory, and does so at the same time as other threads.

The mask can be used, say, to make odd threads inactive, (1), the second half of each warp inactive, (10_{16}), or even alternating warps inactive (20_{16}).

Data from runs with varying masks appears below, notice the new column MASK for the mask value. The first row shows an ordinary run, all threads execute the sine loop. In the second row (first row of the second group), for mask 1, odd-numbered threads do not execute the sine. Notice that it makes little difference. (We would expect more than a 1 or 2% change.) With mask 0x10 threads in lanes 0-15 execute but 16-31 don't. In the third group two mask bits are set, that will enable only 8 threads to be active, subsequent groups show results with 4, 2, and 1 thread active. In all cases it makes no difference.

The last group shows performance when entire warps are suppressed. With mask 0x20 odd-numbered warps are suppressed and finally, we have improvement. But why? We know on this device that there are four warp schedulers. It's possible that schedulers 0 and 1 share two special (sine, etc) functional units, and that schedulers 2 and 3 share another two units. For mask 0x20 schedulers 1 and 3 have no sines, so schedulers 0 and 2 have access to all of the units, improving performance. Pattern 0x40 suppresses execution of warps assigned to schedulers 2 and 3, but not 0 and 1. Schedulers 0 and 1 can't use the functional units assigned to 2 and 3 and so there's no speedup. (The fact that schedulers 2 and 3 aren't doing much work does not make the work assigned to schedulers 0 and 1 go any faster, and so there's no improvement.) Finally, for mask 0x80 the warps assigned to the four schedulers either all have sine operations or none of them do. In that case we'd expect performance improvement.

```
                        MASK
32 wp  64 acwp  100 µs    0     84 GB/s **********************

--- Sixteen threads per warp active.
32 wp  64 acwp  101 µs  0x1     83 GB/s ********************
32 wp  64 acwp  102 µs  0x2     82 GB/s ********************
32 wp  64 acwp  102 µs  0x4     83 GB/s ********************
32 wp  64 acwp  101 µs  0x8     83 GB/s ********************
32 wp  64 acwp  101 µs 0x10     83 GB/s ********************

--- Eight threads per warp active.
32 wp  64 acwp  100 µs  0x3     84 GB/s ********************
32 wp  64 acwp  102 µs 0x18     82 GB/s ********************

--- Four threads per warp active.
32 wp  64 acwp  101 µs  0x7     83 GB/s ********************
32 wp  64 acwp  101 µs 0x1c     83 GB/s ********************

--- Two threads per warp active.
32 wp  64 acwp  101 µs  0xf     83 GB/s ********************
32 wp  64 acwp  101 µs 0x1e     83 GB/s ********************

--- One thread per warp active.
32 wp  64 acwp  102 µs 0x1f     83 GB/s ********************

--- Suppress entire warps in different patterns.
32 wp  64 acwp   77 µs 0x20    109 GB/s ***************************
32 wp  64 acwp  101 µs 0x40     83 GB/s ********************
32 wp  64 acwp   78 µs 0x80    107 GB/s ***************************
```