LSU EE 7700-2 Homework 3 Due: 15 February 2013

Follow the instructions on the class procedures page, `http://www.ece.lsu.edu/gp/proc.html` *for account setup and homework, substituting* `hw3` *for* `hw1` *where appropriate. Also, the file to edit is* `hw3.cu`*, not* `hw3.cc`*. The assignment code is the same as the vertex transformation code used in class.*

Problem 0: Compile and run the homework 3 assignment code unmodified and without command line options. The output will look something like this:

```
Using GPU 0
GPU 0: Tesla C2050 @ 1.15 GHz WITH 2687 MiB GLOBAL MEM
GPU 0: CC: 2.0  MP: 14  TH/WP:  32  TH/BL: 1024
GPU 0: SHARED: 49152  CONST: 65536  # REGS: 32768

CUDA Routine Resource Usage:
 prob1:      0 shared, 112 const, 0 loc, 15 regs; 1024 max threads per block.

Preparing for 1 GPU threads operating on 1048576 vectors of 4 elements.
Launching with 1 blocks of 1 threads.
Elapsed time for 1 threads and 1048576 elements is 306394.012 s
Rate 0.096 GFLOPS,  0.110 GB/s
```

The code starts by printing information about the GPU, see the routine `print_gpu_info` to see how that's done. `MP` is the number of streaming multiprocessors, `TH/WP` is the number of threads per warp, `TH/BL` is the maximum number of threads per block. The next line gives the amount of shared and constant memory, and the number of registers, all per MP.

The line after `CUDA Routine Resource Usage:` describes the GPU code we are running. It gives the amount of static shared memory used, constant memory, local memory, registers, and the maximum number of threads per block for this routine (which can be smaller than the limit for the GPU).

The code will accept three command line arguments. The first is the number of blocks to launch, the second is the number of threads per block, the third is the size of the array in MiB. If the program is run with arguments x y z it will launch with xy threads and an array with $\lfloor z2^{20} \rfloor$ elements. The first two arguments are expected to be integers, the third can be floating point. The default value for each argument is 1. To launch 14 blocks of 256 threads and operate on a 0.5 MiB array use arguments `14 256 0.5`.

Problem 1: Run the program to determine the effect of launch configuration on execution efficiency.

(a) Based on the description of NVIDIA GPUs presented in class we would expect that performance will increase linearly as the number of threads per block increases from 1 to 32, after that performance increases should level off at some point, perhaps at 256 threads per block. Determine if this is true for the code above by running it with different arguments, but always launch with one block.

(b) Determine the difference in performance between using one MP (by setting the block count to 1) and running with multiple MPs. For a small number of threads, say 128, why might one expect the execution rate to be the same if they were all on one MP or spread over multiple MPs?

Problem 2: Modify the program so that it finds the transformed vertex with the smallest magnitude. That is, compute $m = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$ (which is after the transformation) and keep track of the vertex number (value of `h`) with the smallest m. Assign the variables `minimum_mag_index` and `minimum_mag_val` with the correct values, dummy assignments of these variables appear after the commands copying data back from the GPU.

The `App` structure already has a member named `find_minimum_magnitude`, only compute the magnitude when this is true. (This way a solution to this problem won't interfere with problem 1.) The member is set to true when the block count is negative, for example, when run with `-24 256`, the kernel will launch 24 blocks of 256 threads and the `find_minimium_magnitude` member will be set to true. The `hw3` code will check your answer.

Our goal is to have the GPU do most of the work, so don't have the CPU go through all n of the magnitudes (where n is the number of vertices) to find the smallest one. At least each thread should maintain its own minimum, leaving the CPU to look at T items, where T is the number of threads. Shared memory can be used to reduce the number of items to B, where B is the number of blocks, however for this problem the use of shared memory is optional.