LSU EE 7700-2 **Homework 3** Solution **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.

The table below shows the performance for varying block sizes with each row showing the fastest of five runs. This performance does not agree with the expectation of linear speedup mentioned in the problem. For example, if speedup were linear the FP rate at 8 threads would be 8 times the rate for one thread, $8 \times 0.078 = 0.624$, but it's only 0.133 GFLOPS, a bit more than a fifth of what is expected.

Notice also that the performance at block sizes that are a power of 2 (such as 256) is worse than block sizes that are almost the same but not a power of 2. For example, $255 = 3 \times 5 \times 17$ threads is much better than $256 = 2^8$ threads.

```
Blocks   1,  Thds/Blk   1,  Rate  0.078 GFLOPS,  Data  0.089 GB/s
Blocks   1,  Thds/Blk   2,  Rate  0.122 GFLOPS,  Data  0.139 GB/s
Blocks   1,  Thds/Blk   3,  Rate  0.127 GFLOPS,  Data  0.145 GB/s
Blocks   1,  Thds/Blk   4,  Rate  0.170 GFLOPS,  Data  0.194 GB/s
Blocks   1,  Thds/Blk   8,  Rate  0.133 GFLOPS,  Data  0.152 GB/s
Blocks   1,  Thds/Blk  16,  Rate  0.142 GFLOPS,  Data  0.162 GB/s
Blocks   1,  Thds/Blk  32,  Rate  0.263 GFLOPS,  Data  0.301 GB/s
Blocks   1,  Thds/Blk  33,  Rate  0.373 GFLOPS,  Data  0.426 GB/s
Blocks   1,  Thds/Blk  48,  Rate  0.446 GFLOPS,  Data  0.510 GB/s
Blocks   1,  Thds/Blk  49,  Rate  0.550 GFLOPS,  Data  0.629 GB/s
Blocks   1,  Thds/Blk  64,  Rate  0.308 GFLOPS,  Data  0.352 GB/s
Blocks   1,  Thds/Blk  96,  Rate  0.941 GFLOPS,  Data  1.075 GB/s
Blocks   1,  Thds/Blk 128,  Rate  1.037 GFLOPS,  Data  1.185 GB/s
Blocks   1,  Thds/Blk 192,  Rate  1.360 GFLOPS,  Data  1.554 GB/s
Blocks   1,  Thds/Blk 255,  Rate  1.610 GFLOPS,  Data  1.840 GB/s
Blocks   1,  Thds/Blk 256,  Rate  1.262 GFLOPS,  Data  1.442 GB/s
```

(*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?

On a Quadro 2000M the data shows increasing performance with a larger number of blocks, up to 32 blocks.

Each multiprocessor can execute a certain number of operations per cycle. On the CC 2.1 device for which the data in this assignment was collected, that is 48 single-precision floating-point operations per clock cycle (the clock period is $0.909\,\text{ns}$). Compare a launch of 1 block of 16 threads with 2 blocks of 8 threads. In the 1-block configuration all threads are on one multiprocessor but there are still enough units so that a thread will not have to wait for other threads to finish. Therefore, *as far as the FP rate goes*, there is no advantage in launching with 2 blocks.

Consider next, a large number of threads, say 1024. With that many threads, in a 1-block configuration a thread would have to wait for another thread to finish since there are only 48 FP units for 1024 threads. Because a thread has to wait for a FP unit, the FP units are always busy and so the MP will execute at its peak rate of 48 FP per cycle. In a launch with 2 blocks of 512 threads each, the block scheduler will put each block on its own multiprocessor and so a total of 96 FP units (CUDA cores) will be available, for a peak of 96 FP operations per cycle, which is twice as high.

So is 128 threads more like the 16-thread case, in which no speedup is expected, or the 1024-thread case, where linear speedup is expected? That question can't be answered without looking for the code, but in class it was suggested that for many CUDA kernels the benefit of increasing the block size starts to level off at 256 threads. So for 128 threads we would still expect FP units to be idle.

If that's true, why might have performance improved? Probably because of cache size. The code example makes poor use of the cache and memory system (as will be covered in class), so having more cache helps it. The right fix, though is to change access order, see the classroom examples.

```
Blocks   1,  Thds/Blk 128,  Rate  1.074 GFLOPS,  Data  1.227 GB/s
Blocks   2,  Thds/Blk  64,  Rate  1.034 GFLOPS,  Data  1.182 GB/s
Blocks   4,  Thds/Blk  32,  Rate  1.408 GFLOPS,  Data  1.609 GB/s
Blocks   8,  Thds/Blk  16,  Rate  1.875 GFLOPS,  Data  2.143 GB/s
Blocks  16,  Thds/Blk   8,  Rate  2.815 GFLOPS,  Data  3.217 GB/s
Blocks  32,  Thds/Blk   4,  Rate  4.041 GFLOPS,  Data  4.618 GB/s
Blocks  64,  Thds/Blk   2,  Rate  3.338 GFLOPS,  Data  3.815 GB/s
Blocks 128,  Thds/Blk   1,  Rate  2.315 GFLOPS,  Data  2.646 GB/s
```

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.

The solution has been checked into the repo with a file name of `hw3-sol.cu`.