LSU EE 7700-2 Homework 6 Due: 27 March 2013

Follow the instructions on the class procedures page, `http://www.ece.lsu.edu/gp/proc.html` *for account setup and homework, substituting* `hw6` *for* `hw1` *where appropriate. Also, the file to edit is* `hw4.cu`*, not* `hw6.cc`*. The assignment code is the same as the vertex transformation code used in class.*

Problem 0: Read the background provided in this problem, and compile and run the Homework 6 assignment code; report any problems as soon as possible. The answer to this problem is "It ran fine.", the next problem actually asks questions.

The program runs four GPU versions of our vertex transformation code, `kernel_many_threads`, `kernel_few_threads_d2`, `kernel_few_threads_d4`, and `kernel_few_threads_d8`, set to operate on 2-element vertices.

The code for his assignment is similar to that used in Homework 4. One important change is the forcing of kernels to use a single load instruction to load a vector, rather than one load for each element. (Using four loads to load a 4-element vector can result in re-fetching a line to the L1 cache due to conflict misses.)

Kernel `kernel_many_threads` is similar to the kernel is Homework 4. The other kernels are hand-unrolled versions of `kernel_many_thread`. For example, `kernel_few_threads_d4` contains four copies of the loop body. The unrolling was performed in such a way that all loads in an iteration would be performed before any calculation. The template `kernel_few_threads_d` contains the actual code, `kernel_few_threads_d4` instantiates the template for a degree of 4, etc.

There are two command line arguments, the first is the number of blocks to launch, the second is the size of the array to use, in MiB, (fractional amounts are fine).

The program will launch each kernel in up to 32 configurations, from a 1 warp-block to the maximum block size possible for the kernel. For each launch the kernel execution time and data transfer rate are shown. The data transfer rate is shown in GB/s, as a percentage of the maximum GPU to device memory bandwidth, and as a bar graph.

Note that the number of blocks must be appropriately chosen for some of the problems below.

Problem 1: The inspiration for this assignment is the choice between many-thread systems (like GPUs) and fewer-thread systems (like the so-called manycores). Kernel `kernel_many_threads` is written to rely on a large number of threads to hide latency. The other kernels rely on loop unrolling to hide at least some latency, and so fewer threads will be required, however those threads will use more registers because the compiler is putting greater distance between the instruction that writes a register and the first instruction that uses the register.

(*a*) Perform a set of runs (or just one run if that's all it takes) to determine which is the more efficient approach in terms of the amount of hardware needed. For each kernel indicate the number of threads needed to realize something close to the best performance and also indicate the **total** number of registers needed. (That is, the number of registers the multiprocessor needs in order to run that number of threads for that kernel.) Provide relevant information such as the type of GPU, and number of blocks chosen for each experiment, etc.

Do this for two goals: one to achieve the fastest computation (which should saturate off-chip data bandwidth) and one to achieve fastest performance on at least one multiprocessor. Note that for the first goal memory latency will not be the performance limiter and so it is not necessary to cover all memory latency to realize maximum performance.

(*b*) Comment on whether these experiments allow one to conclude that the many-threads or fewer-threads approach is better. An answer might start "These experiments don't show the full benefit of the fewer-threads approach because the code in `kernel_few_threads_d` ..."

Problem 2: Perform the following hand analysis of kernels `kernel_many_threads` and `kernel_few_threads_d4`. Locate the assembler code (in the file ending with `sass`) and identify the loop body for each routine (for routine `kernel_few_threads_d4` do the unrolled loop, not the loop at the end).

Estimate the latency of the loop body under the assumption that memory instructions have a 400-cycle latency and all others have a 24-cycle latency.

Here are some miscellaneous facts: Instructions with a `.X` completer depend on the most recent instruction with a `.CC` completer. Sixty-four bit load instructions load a pair of registers, the named destination is only the first of two. For example, this load `LD.E.64 R2, [R8];` instruction loads both register `R2` and `R3`. The 128-bit load instructions load four registers.

To find the latency of the loop body (a single iteration) determine the latest time that an instruction will start. For convenience assume a device of CC 2.0, so that instruction $i + 1$ starts two cycles after instruction i when it is not dependent on prior instructions. A dependent instruction starts when its operands have been computed based on the start time and the latencies given above.

In the example below, instruction 0 starts at time 0 and its result is ready at time 24. Instruction 1 starts at 2 since it's not dependent on 0. However, 2 depends on 0 because of `R10` and so it must wait until 24 to start. Instruction 3 depends on instruction 2 through the carry bit (the `.CC` to `.X` dependence). Instruction 4 can start at cycle 50 since by then its operands will be available.

```
0: MOV32I R10, 0x8;           // Start at 0, result ready at 24
1: IMUL.HI R4, R0, 0x8;       // Start at 2, result ready at 26
2: IMAD R8.CC, R0, R10, c [0x2] [0x30]; // Start at 24 (dep on 1st insn via R10) rdy at 48
3: IADD.X R9, R4, c [0x2] [0x34];       // Start at 48 (dep on prev via CC) rdy at 72
4: IMAD R6.CC, R0, R10, c [0x2] [0x40]; // Start at 50, result ready at 74
...
```

Use the following method to help find dependencies. In Emacs put the cursor over the start of a destination register, perhaps the first `R10` in the example above. Then press C-s C-w (control-s followed by a control w). This should highlight all occurrences of R10. (The C-s starts a search, and C-w tells Emacs to search for other occurrences of the word under the cursor. A second C-s would move to the next occurrence.) Press C-g to exit the search.

(*a*) Show the latency of the loop body of each kernel, as described above.

(*b*) Count the number of instructions in the loop bodies.

(*c*) Based on the answer to the last two parts, determine the minimum number of warps per multiprocessor needed to make full use of the CUDA cores on a device of compute capability 2.0. Assume unlimited data bandwidth (but global load latency is still 400 cycles). Note that this is a lower bound on the number of warps needed to completely hide latency.

(*d*) Using data from the device used in the previous problem, determine the minimum number of warps per multiprocessor needed to make full use of the memory bandwidth.

(*e*) Compare these answers to the experiments performed in the previous problem. Comment on how closely they agree.