**LSU EE 7700-2** **Homework 4** Solution **Due: 26 February 2013**

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

**Problem 1:** Compile and run the Homework 4 assignment code. This code is similar to that used in Homework 3 except rather than finding the minimum vertex magnitude, it finds the sum of all the squared magnitudes. This makes the code simpler, and also makes the error-checking code more sensitive to errors (which is a good thing).

The first command-line option specifies the method of finding the global sum (a sum is an example of a reduction). The code currently works with methods 0 to 3, in the next problem method 4 will be implemented.

The program will accept three other command line arguments, those match the previous assignment. The first argument is the reduction method to use, the second is the number of blocks to launch, the thrid is the number of threads per block, the fourth is the size of the array in MiB. If the program is run with arguments $r$ $x$ $y$ $z$ it will launch with $xy$ threads and an array with $\lfloor z2^{20} \rfloor$ elements, and use reduction method $r$ to find a global sum.

As with the prior assignment, the timing is of the GPU portion only. When comparing timings, note that with reduction method 0 the GPU does not perform any reduction, so it will appear to be faster than the other methods (since the CPU time isn't being counted).

Run the program with methods 0 to 3. Prepare a table with a row for each method. Show the run time and the following other information: the number of synchronizations performed per thread, and the number of additions performed to find the global sum by the *critical thread*. The *critical thread* is the thread that performs the most work. In many of our examples, that will be thread 0.

Solution on next page.

The table appears below, including the data from Problem 3. The data was collected for a Tesla C2050 which has 14 multiprocessors and implements CC 2.0. The code was run at different grid sizes to vary the impact of reduction on execution time.

The total number of vertex transformations is the same regardless of the number of blocks. However, the number of reductions is equal to the number of blocks; the number of reductions per multiprocessor appears in the last column. See the Problem 3 solution for a discussion of this data.

SOLUTION

| Mtd | Grid Size | Block Size | GPU Time Microsec | Syncs / Blk | Additions / Blk | Reductions / MP |
|---|---|---|---|---|---|---|
| 0 | 14 | 1024 | 383 | 0 | 0 | 1 |
| 1 | 14 | 1024 | 414 | 1 | 1023 | 1 |
| 2 | 14 | 1024 | 386 | 10 | 10 | 1 |
| 3 | 14 | 1024 | 384 | 1 | 31 + 5 = 36 | 1 |
| 4 | 14 | 1024 | 383 | 1 | 5 + 5 = 10 | 1 |
| 0 | 224 | 1024 | 361 | 0 | 0 | 16 |
| 1 | 224 | 1024 | 717 | 1 | 1023 | 16 |
| 2 | 224 | 1024 | 357 | 10 | 10 | 16 |
| 3 | 224 | 1024 | 354 | 1 | 36 | 16 |
| 4 | 224 | 1024 | 352 | 1 | 10 | 16 |
| 0 | 1024 | 1024 | 353 | 0 | 0 | 73 |
| 1 | 1024 | 1024 | 2573 | 1 | 1023 | 73 |
| 2 | 1024 | 1024 | 521 | 10 | 10 | 73 |
| 3 | 1024 | 1024 | 427 | 1 | 36 | 73 |
| 4 | 1024 | 1024 | 390 | 1 | 10 | 73 |
| 0 | 4096 | 256 | 374 | 0 | 0 | 292 |
| 1 | 4096 | 256 | 713 | 1 | 256 | 292 |
| 2 | 4096 | 256 | 364 | 8 | 8 | 292 |
| 3 | 4096 | 256 | 314 | 1 | 13 | 292 |
| 4 | 4096 | 256 | 313 | 1 | 10 | 292 |

**Problem 2:** Locate the routine `reduction_method_4`. Add code to this routine so that the reduction is performed by using two tree reductions. In the first reduction each warp will do its own reduction. That is, after the first reduction we'll have the sum of threads 0-31, a separate sum for 32-63, etc. If the block size were 1024 threads (the maximum block size in a Fermi device) we would have 32 separate sums. Use the second reduction to find the sum of these 32 (or fewer) sums.

This can be written so that only one synchronization is needed, between the two reductions.

The solution code checked into the repository in file name `hw4-sol.cu`. An htmlized version posted at `http://www.ece.lsu.edu/gp/2013/hw4-sol.cu.html`.

**Problem 3:** Add the data for your reduction routine to the table. Comment on the performance of each method. Which do you think should be used?

If our goal is to evaluate which reduction method is fastest we should run configurations with the maximum number of threads and with the fewest vertices per thread. Since the default array size is $2^{20}$ that would be a block size of 1024

(the maximum for CC 2.0) and a grid size of 1024 blocks. With this configuration each thread transforms just one vertex and then performs a reduction. (In contrast, at block size 10 each thread transforms $2^{20}/(10 \times 1024) \approx 102$ vertices.)

At this configuration Method 1 is clearly the worst, as one would expect since a single thread does all the work on a multiprocessor, using at most $1/32$ of the compute capability. Method 2, the full tree reduction, performs fewer adds but more synchronizations than Method 3, which avoids synchronization by using only one warp for reduction. Based on their performance, reducing the synchronizations by a factor of ten improves performance enough to make up for having 3.6 times as many additions. Method 4 avoids both the synchronizations and the extra additions, and so yields the best execution time.

At a grid size of 14 the differences between the reduction methods are much smaller since fewer reductions are being performed: just one per MP, versus $\frac{1024}{14} \approx 73$ per MP at a grid size 1024.

With a smaller block size, 256 threads, the advantage of Method 4 over Method 3 shrinks because the linear sum portion of Method 3 iterates $256/32 = 8$ times (compared to 32 for a 1024-thread block). Note the solution code for Method 4 will perform 10 reduction steps regardless of the block size. That yields better performance at larger block sizes because the number of iterations in the second reduction loop is a compile-time constant.

The smaller block sizes with the better reduction methods outperform the larger ones. This may be due to higher warp occupancy.