LSU EE 7700-2 **Homework 5** Solution **Due: 13 March 2013**

For details on how the GPU works see the CUDA C Programming Guide, linked to `http://www.ece.lsu.edu/gp/gp/ref.html`. For hardware details in particular see Chapter 1 (brief history and overview), Chapter 4 (organization overview), and Sections 5.2.3 (warp/instruction scheduling), 5.3.2 (memory), 5.4 (instruction scheduling and functional units), and Appendix F (some more details on resources and instruction handling).

Problem 1: The Volkov paper describes algorithms for dense linear algebra on GPUs, it is linked to `http://www.ece.lsu.edu/gp/gp/ref.html` under the heading "Matrix Multiplication." The paper is important as much for the algorithms themselves, as for the methodical approach used in characterizing the GPUs and developing the algorithms. The paper was written for the NVIDIA CC 1.x generation of GPUs, which are now obsolete. Nevertheless, the paper is valuable in showing how to measure the capabilities of a device and tune code for it.

Read the following sections of the Volkov paper: Abstract, Introduction, Section 2, 2.1, 2.3, and Section 3.

(*a*) Add a column to Table 1 for the following devices: A GTX 580 and a Kepler K20.

Transposed column appears below. Notes: What the paper calls a core, NVIDIA calls a multiprocessor. The peak computation rate counts a multiply/add instruction as two floating-point operations. So, for example, the GTX single-precision rate is $16 \times 32 \times 1544 \times 2 = 1581056$.

SOLUTION:

Device Name	Num cores	Core clock MHz	Reg/ core	smem/ core	Bus GHz	Bus pins	Bus GB/s	Mem MiB
GTX 580	16	1544	128 kiB	48 kiB	2004	384	192.4	1536
Kepler K20	13	710	256 kiB	48 kiB	2600	320	208.0	5000

Device Name	SP Peak per GPU GFLOPS	SP Peak per core GFLOPS	SP FLOPS per word	DP Peak per GPU	DP FLOPS per word
GTX 580	1581	98.8	32.9	790*	32.9
Kepler K20	3544	272.6	68.1	1181	45.4

* Actually ‘‘lower’’ according to C Programming Guide

As mentioned in class, NVIDIA GPUs have high instruction latency, but are designed so that the latency can be hidden by using lots of threads.

(*b*) Find the part of the paper in which Volkov describes the experiments used to find instruction latency. What latencies does he report?

Section 3.4. For the GTX 280 they report 24 cycles for ordinary single-precision instructions that operate solely on registers, slightly longer when one operand is from shared memory. Special (such as reciprocal square root) take 28 cycles, DP add/mult take 48 cycles, DP fma take 52 cycles.

(*c*) This section also gives the number of warps (warps aren't called warps elsewhere in the paper) needed to hide this latency. Show how that number is computed.

They find that 6 warps are sufficient to cover the 24-cycle latency. The paper doesn't explicitly explain how the number 6 is computed.

(*d*) Find the place in the NVIDIA documentation which describes the number of warps needed to hide latency. Show the number of warps for each generation of CUDA architecture (1.x, 2.0, 2.1, 3.x) needed to hide latency assuming code consisted of just add and multiply instructions (which is sort of the default case). Also show the warps needed under the assumption that the code consists of reciprocal instructions. (Note that this part does not require the Volkov paper.)

This is discussed in section 5.2.3 of the CUDA C Programming Guide (v5.0). The number of warps needed to hide a latency of L clock cycles is given as $L/4$ for CC 1.x, L for CC 2.0, $2L$ for CC 2.1, and $8L$ for CC 3.X. To hide the latency of a single-precision multiply or add instruction, 24 cycles on CC 1.X and CC 2.X and 12 cycles on CC 3.X, one would need 6 warps for CC 1.X, 24 warps for CC 2.0. Because of dual (superscalar) issue it is probably impossible to completely hide the latency of two dependent instructions when one immediately follows the other since the scheduler will need two independent instructions to make use of all 48 CUDA cores. Ignoring this fact it would take 48 warps for CC 2.1, and an unachievable 96 warps for CC 3.0.

Note that the numbers for CC 2.1 and 3.X don't agree with the analyses presented in class. For CC 2.1, if instructions were initiated for $2L$ warps then that would require 64 CUDA cores per cycle, but the device has only 48.

If code consists of reciprocal instructions, then it would take longer to issue each warp. The number of cycles is $32/2 = 16$ for CC 1.X, $32/4 = 8$ for CC 2.0, $32/8 = 4$ for CC 2.1, and $32/32 = 1$ for CC 3.x. The number of warps needed to hide a latency of L cycles when the code consists of reciprocal instructions is then $L/16$ for CC 1.x, $L/8$ for CC 2.0, $L/4$ for CC 2.1, and L for CC 3.x.

(*e*) The impact of instruction latency is not as bad if the instruction reading a register does not immediately follow the instruction writing the register. Find the part of the paper that verifies that this is indeed true. What is the minimum number of warps necessary to avoid stalls on code with suitably distant dependencies, according to the paper?

This is discussed in Section 3.6. They get close to peak throughput with just two warps on code with distant dependencies (made possible by unrolling the loop).

Problem 2: We know that the `synchthreads()` call should be avoided because it adds overhead. Shown below is an excerpt of the machine code that performs a tree reduction of values within a warp, each group of three instructions adds on a value at a different distance. Notice that each instruction is dependent on the instruction before it.

```
@!P4 LDS R3, [R2+0x40];
@!P4 FADD R0, R0, R3;
@!P4 STS [R2], R0;

@!P0 LDS R3, [R2+0x20];
@!P0 FADD R0, R0, R3;
@!P0 STS [R2], R0;

@!P1 LDS R3, [R2+0x10];
@!P1 FADD R0, R0, R3;
@!P1 STS [R2], R0;

@!P2 LDS R3, [R2+0x8];
@!P2 FADD R0, R0, R3;
@!P2 STS [R2], R0;

@!P3 LDS R3, [R2+0x4];
@!P3 FADD R0, R0, R3;
@!P3 STS [R2], R0;
```

(*a*) Compute the execution time of this code, measured in cycles, on a CC 2.0 device, for the launch of a single block with 1024 threads. Assume that all instruction latencies are 24 cycles. Measure time from the execution of the first instruction (shown as something like $\boxed{\text{I0}}$ in class) to the execution of the last instruction.

In a CC 2.0 device, in which there are 32 cuda cores, 24 warps are needed to hide the 24 cycles of latency in the worst case, so all latency is hidden. There are at total of $15 \times 1024 = 15360$ instructions. Assuming shared load and stores can use the 32 cuda cores, the execution time will be $\frac{15360}{32} = 480$ cycles.

(*b*) The code above did not have `synchthreads` between each group. In this part consider code in which there is:

```
@!P4 LDS R3, [R2+0x40];
@!P4 FADD R0, R0, R3;
@!P4 STS [R2], R0;
BAR.RED.POPC RZ, RZ;
@!P0 LDS R3, [R2+0x20];
@!P0 FADD R0, R0, R3;
@!P0 STS [R2], R0;
BAR.RED.POPC RZ, RZ;
@!P1 LDS R3, [R2+0x10];
@!P1 FADD R0, R0, R3;
@!P1 STS [R2], R0;
BAR.RED.POPC RZ, RZ;
@!P2 LDS R3, [R2+0x8];
@!P2 FADD R0, R0, R3;
@!P2 STS [R2], R0;
BAR.RED.POPC RZ, RZ;
@!P3 LDS R3, [R2+0x4];
@!P3 FADD R0, R0, R3;
@!P3 STS [R2], R0;
BAR.RED.POPC RZ, RZ;
```

Assume that the barrier instruction, `BAR`, uses cuda cores for execution (as does the `FADD`), and that its latency is 24 cycles. An instruction following a barrier cannot execute until 24 cycles after *the last thread* executes its barrier instruction. (In contrast, the `FADD` for a thread cannot execute until at least 24 cycles after the `LDS` in the same thread executes.)

Compute the time for the code above for this assumed behavior.

The BAR instruction breaks execution into five sections. The first section has four instructions (including the BAR), its execution time is $\frac{4\times1024}{32} = 128$ cycles. Taken alone, each of the other sections has the same execution time. Because of the barrier, the second section cannot start execution until 24 cycles after the end of the first, therefore the total time is $5 \times (128 + 24) = 760$ cycles. This is longer than the $\frac{5\times4\times1024}{32} = 640$ cycles that would be achieved if the barrier did not force instructions to wait.