LSU EE 7722 Homework 1 Due: 28 February 2014

Problem 0: If necessary, follow the instructions for class account setup and homework workflow on the course procedures page `http://www.ece.lsu.edu/gp/proc.html`.

For background and help with CUDA programming see the CUDA C Programming Guide, a version is linked to the course Web pages at `http://www.ece.lsu.edu/gp/ref.html` it can also be found under directory `/usr/local/cuda/doc` on a system with CUDA installed.

The assignment file, `hw01.cu`, contains three versions of a kernel for computing a simple operation on elements of an array. The first kernel `dots_iterate0`, is written in a straightforward way but its performance suffers because the compiler can't or won't unroll the loop. The second kernel, `dots_iterate1`, is written so that the compiler can unroll the loop, but not in the most efficient way. The third kernel, `dots_iterate2`, initially does nothing, but it does have a copy of the code from `dots_iterate2`. It will be modified as part of the solution to a problem in the next homework assignment.

Compile and run the program. The makefile will generate ptx and CUDA assembly language versions of the CUDA code, in files `hw01.ptx` and `hw01.sass`.

The program starts by writing information about the GPUs that are available, it will always use GPU 0.

```
GPU 0: Tesla K20c @ 0.71 GHz WITH 5119 MiB GLOBAL MEM
GPU 0: CC: 3.5  MP: 13  CC/MP: 192  TH/BL: 1024
GPU 0: SHARED: 49152 B  CONST: 65536 B  # REGS: 65536
GPU 0: L2: 1280 kiB   MEM to L2: 208.0 GB/s  SP 1760.9 GFLOPS  OP/ELT 33.86
GPU 1: Quadro 5000 @ 1.03 GHz WITH 2559 MiB GLOBAL MEM
GPU 1: CC: 2.0  MP: 11  CC/MP:  32  TH/BL: 1024
GPU 1: SHARED: 49152 B  CONST: 65536 B  # REGS: 32768
GPU 1: L2: 640 kiB   MEM to L2: 120.0 GB/s  SP 361.2 GFLOPS  OP/ELT 12.04
Using GPU 0
```

Next it shows the resources used by each of the three kernels. In this assignment we will be looking at the number of registers (`regs`) and the amount of local storage needed per thread (`loc`).

```
CUDA Kernel Resource Usage:
For dots_iterate0:
       0 shared, 64 const, 0 loc, 13 regs; 1024 max threads per block.
For dots_iterate1:
       0 shared, 64 const, 0 loc, 38 regs; 1024 max threads per block.
For dots_iterate2:
       0 shared, 64 const, 0 loc, 2 regs; 1024 max threads per block.
```

After printing this information the program runs each kernel multiple times, in a launch configuration varying from 1 to the maximum number of warps. (The sample below does not show the entire output.)

```
Preparing for 13 blocks operating on 1048576 elements. Unroll degree: 8.

Running kernel dots_iterate0 which uses 13 regs on 13 blocks.
 Num      Time    Data
 Wps         s    GB/s    Pct
  1   1434.976    8.77   4.2%  000
  2    726.944   17.31   8.3%  00000
```

```
3     489.984    25.68  12.3%  0000000
4     371.008    33.92  16.3%  00000000
5     299.936    41.95  20.2%  0000000000
6     254.528    49.44  23.8%  000000000000
7     219.712    57.27  27.5%  00000000000000
8     195.712    64.29  30.9%  0000000000000000
```

The timing for each configuration is printed, along with the percentage of peak data bandwidth realized by the code. The peak bandwidth is shown as a number and as a line of digits (zeros above). If the percentage, under `Pct`, is 100% that means the kernel is transferring data at the maximum rate (based upon certain assumptions).

Before each kernel is run the program prints the number of registers used by the kernel (per thread) and, if applicable, the loop unrolling degree. (The first line in the sample above.)

If there are any difficulties running the code ask for help. There is nothing to submit for this first problem.

Problem 1: Examine the code for kernel `dots_iterate0`.

(*a*) Assume that the address of array `a` starts at `0x1000` and consider a system with 12 multiprocessors and a launch configuration of 12 blocks of 256 threads each. For block 8, threads 0, 1, and 2 find the address of a[idx].x for the first two iterations of the loop. That is, compute six memory addresses. Do this by hand and show your work.

(*b*) As we should already know, memory access is most efficient when the loads issued by the threads in a half-warp access a contiguous area of memory. Consider the two accesses to `a` made in each iteration of the loop in `dots_iterate0` and in the other kernels for this assignment:

```
b[idx] = dapp.v0 + dapp.v1 * a[idx].x + dapp.v2 * a[idx].y;
```

Explain why the requests for `a` would only be contiguous if the compiler recognizes something about the two accesses to `a` and if an appropriate load instruction exists. To answer this question think about the data type of `a`.

Problem 2: Recall that in loop unrolling, a loop that handles, say, one array element per iteration and performs n iterations is replaced with a loop that handles d elements per iteration and which performs $\lceil N/d \rceil$ iterations, d is called the *degree*. For example the loop

```
for ( int idx = 0; idx < 16; idx++ )
  b[idx] = v0 + v1 * a[idx];
```

can be replaced by

```
for ( int idx = 0; idx < 16; idx += 2 ) {
    b[idx] = v0 + v1 * a[idx];
    b[idx+1] = v0 + v1 * a[idx+1]; }
```

in a degree-2 unrolling assuming the number of iterations is even. If the number of iterations is not a multiple of the degree then extra *prologue* or *epilogue* code can be added before or after the loop to handle the $n \bmod d$ extra iterations.

Loop unrolling results in code with fewer instructions and better-scheduled instructions—if done properly. There are fewer instructions for several reasons. Without unrolling the code that checks for the end of the loop executes n times, with a degree-d unrolling it executes n/d times. Consider `b[idx]` and `b[idx+1]` in the example above. Without unrolling the address of `b[idx]` would have to be computed n times.

(The address is `b + idx * 4` , where `b` is the address of the start of the array and the size of an array element is 4 bytes. For this example, the compiler would use a left shift rather than a multiply by 4. If might also add 4 to the address computed in the previous iteration.)

With unrolling the compiler may only need to emit code to compute the address for `b[idx]`, not for `b[idx+1]`, that is because certain load instructions can add constants, called *offsets*, to a base address. When that is the case, the time needed for address computation is also reduced. Offsets are available in most CPU load instructions. For NVIDIA CC 2.X and CC 3.X offsets are available for instructions that don't access the texture cache. This will be the case for the assignment code whether run on Fermi or Kepler systems.

Unrolling also provides the compiler more freedom to schedule (move around) instructions. With this freedom the compiler can place loads far away from instructions that use the loaded values.

The NVIDIA compiler will unroll loops without being told to do so if the number of iterations is known at compile time. That is not the case in `dots_iterate0`, since the number of iterations is based on variables that can be assigned different values at run time. In contrast the number of iterations of the `i` loops in `dots_iterate1` is `unroll_degree` which is a constant known at compile time (it is set at the top of the file). Therefore the compiler will unroll these loops. (A pragma is available to tell the compiler to unroll a loop even if the iteration count is unknown.)

One benefit of unrolling is providing the compiler the freedom to schedule load instructions far from their uses. However, when the compiler moves instructions it must take care to not change what the program does (that is, it can't compute the wrong answer). That is, it cannot move instruction A before instruction B if B computes a result that A needs. This creates a dilemma for the compiler when instructions load and store from memory. In the general case the compiler has no way of knowing if some load instruction loads a value from the same address that a store instruction stores to. If two such instructions refer to the same address their addresses (or pointers) are said to *alias*. If the compiler were to move the load of such a pair before the store, the program would be incorrect.

Because of this conservatism on the part of the compiler a major benefit of unrolling would not be realized when unrolling the code in `dots_iterate0` because it can't rule out that, say, `b[0]` aliases with (has the same address as) `a[thread_count].x`.

Because the threat of aliasing is a big impediment to the generation of good code, many compilers provide options to ignore the threat of aliasing. However, gcc and nvcc (the NVIDIA compiler) have no such options. These compilers do recognize the `__restrict__` variable attribute that tells the compiler that aliasing will not occur. However nvcc will ignore `__restrict__` under many circumstances.

Because nvcc code is so picky about the restrict attribute the homework assignment code does not use it. Routines `dots_iterate1` avoids the need for restrict by temporarily writing values destined to memory in an array named `keep`, that is done by the first `i` loop. The second `i` loop writes these values to memory.

The code in `dots_iterate1` has been written in a way to enable efficient unrolling. Because the iteration count of the two `i` loops is a compile-time constant the compiler will unroll them. Because writes to global memory are done only in the second `i` loop the compiler can safely schedule the loads in the first `i` loop.

(*a*) Looking at the performance of configurations of `dots_iterate0` with different number of warps, find the one with shortest run time, note the number of warps needed to achieve it and call the run time t_0. Next, run the code with different values of `unroll_degree`. (To do this you'll need to modify `unroll_degree` in the assignment file, recompile the code, and then re-run it.) Find the value of `unroll_degree` that achieves a run time of t_0 or less using the fewest number of registers *per block*. The number of registers per block is the product of the number of registers used by the kernel (it is printed when the program runs) and the number of threads in a block.

Indicate the unroll degree and the number of registers used, and compare that to the number of registers used by `dots_iterate0`.

(*b*) The array `keep` is in the local address space. The code is written in such a way that the compiler will use registers rather than load and store instructions for accesses to `keep` if `unroll_degree` is small enough. At unroll degree 64 the compiler will be forced to resort to local memory for accesses to `keep`.

For unroll degree 64 compute the maximum number of warps for which local accesses will be contained in the L1 cache, assuming that nothing else uses the L1 cache. Assume that every reference to `keep` uses a load or store instruction rather than a register.

Set the unroll degree to 64 and observe the results. Run the code and determine whether the performance versus the number of warps agrees with what one would predict based on the answer above. That is, does performance suffer much when the size of the local address space used exceeds the L1 cache size?

(*c*) In Fermi devices both local and global accesses use the L1 cache (be default). Are the accesses to `a` in `dots_iterate0` helped by the L1 cache? Explain.