LSU EE 7722 **Homework 2** **Due: 7 March 2014**

PRELIMINARY, MAY BE MODIFIED

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: The code in `dots_iterate1` is written in a way that enables the compiler to unroll it (by performing the stores after the loads). (See the discussion of loop unrolling in Homework 1.)

Notice that `idx2` is computed in terms of `thread_count`, and that thread count is not known at compile time. For the sake of unrolling it would be much better if the expression for `idx2` was `idx + i * C`, where `C` was a compile-time constant.

(*a*) Modify routine `dots_iterate2` (which initially is a copy of `dots_iterate1`) so that `idx2` is computed using a constant. **Do not** try to make the thread count a compile time constant, use some other strategy.

The array `b` written by the kernels is checked for correctness, so if you make a mistake an error message about a wrong result will be printed.

(*b*) Another barrier to high performance is the `if` statements in the `i` loops. The `if` statements are needed to prevent reads and writes of elements beyond the end of the arrays. One way of avoiding the need for the `if` statements is by making the arrays a little larger. The arrays already are made a little larger, by 256 extra elements. In `dots_iterate2` remove the `if` statements inside the `i` loops. Then, look for where the variable `overrun_size` is assigned, and assign it the smallest value for your code.

(*c*) Describe the benefit of your code in terms of its performance and in terms of the number of registers used. Remember that the number of registers should be based on the number of threads that are needed, not just the number of registers used by a single thread.