LSU EE 7700-2 Homework 1 Solution Due: 12 March 2012

For the following assignment read Chapter 2 in the CUDA C Programming Guide, which is linked to the NVIDIA documentation page, `http://developer.nvidia.com/nvidia-gpu-computing-documentation`, and which can also be found in /usr/local/cuda/doc in a standard Linux CUDA toolkit installation. Also refer to the documentation for `cuobjdump`, which describes the NVIDIA machine language and to the class notes on the GF3 at `http://www.ece.lsu.edu/gp/notes/set-study-gf3.pdf`.

Source code and accounts for this assignment may be made available.

Problem 1: A CUDA program, a variation on the "dots" demo shown in class, operates on a 1000000-element array using a kernel which operates on a single element. (See the code fragments below.)

(*a*) Suppose the block size is set to 64. How many blocks should be launched? Modify the code below so that the kernel is launched with this block size and number of blocks. Note that only an assignment for the `x` component of the grid configuration is shown; add the others. *Note: This is an easy problem, and something close to the solution can be found in other examples. Please solve it on your own.*

We need one thread per array element. It just so happens that 64 is a factor of 1000000, so we need to launch exactly 15625 blocks.

```
__host__ void dots_launch() {
 dim3 dg, db;

 dg.x = 15625;  dg.y = dg.z = 1;
 db.x = 64; db.y = db.z = 1;

  dots<<<dg,db>>>();
}

__global__ void dots() {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if ( idx >= array_size ) return;
  b[idx] = v0 + v1 * a[idx].x + v2 * a[idx].y;
}
```

(*b*) Suppose that the code above is run on a Tesla C2050. How many blocks will there be per multiprocessor? *Note: To solve this problem you need to look at characteristics of the C2050 in the Programming Guide.*

The C2050 has 14 multiprocessors and so there will be $15625/14 = 1116.07$ blocks per multiprocessor. (Some will get more, some will get fewer, depending on the block scheduler.)

(*c*) Suppose the kernel (`dots`) consists of 21 instructions. How many cycles would it take for the kernel to finish on a Tesla C2050 assuming that:

- No instruction has to wait, even for loads.
- All instructions use CUDA cores. (None of them use the special functional units).

Consider an MP that gets 1117 blocks or 71488 threads. It would need to execute 1501248 instructions, with 32 CUDA cores per multiprocessor it would take 46914 cycles to issue them (and 23 more cycles for the last one to finish).

Problem 2: The `dots` kernel and the NVIDIA CC 2.0 machine code for the `dots` kernel appears below.

```
__global__ void dots() {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if ( idx >= array_size ) return;
  b[idx] = v0 + v1 * a[idx].x + v2 * a[idx].y; }
```

```
        code for sm_20 Function : _Z4dotsv
                                              // SOLUTION
/*0000*/ MOV R1, c [0x1] [0x100];
/*0008*/ S2R R2, SR_Tid_X;                     // Get threadIdx.
/*0010*/ S2R R0, SR_CTAid_X;                   // Get blockIdx.x
/*0018*/ IMAD R0, R0, c [0x0] [0x8], R2;       // Compute idx
         // (Below) Set predicate to idx < array size.
/*0020*/ ISETP.LT.AND P0, pt, R0, c [0x2] [0xc], pt;
/*0028*/ @!P0 BRA.U 0xa0;                      // Jump to end if false.
/*0030*/ @P0 MOV R3, c [0x2] [0x10];           // Load address of a[0]
/*0038*/ @P0 IMUL.HI R2, R0, 0x8;              // Multiply idx by a elt size.
/*0040*/ @P0 MOV R6, c [0x2] [0x18];
/*0048*/ @P0 IMAD R8.CC, R0, 0x8, R3;  // Compute low 32 bits of &a[idx]
/*0050*/ @P0 MOV R5, c [0x2] [0x4];            // Load v1
/*0058*/ @P0 IMUL.HI R4, R0, 0x4;              // Multiply idx by b elt size.
/*0060*/ @P0 IADD.X R9, R2, c [0x2] [0x14];  // Compute high 32 bits of &a[idx]
/*0068*/ @P0 IMAD R6.CC, R0, 0x4, R6;        // Compute low 32 b of &b[idx]
/*0070*/ @P0 LD.E R2, [R8];                    // Load a[idx].x
/*0078*/ @P0 LD.E R3, [R8+0x4];                // Load a[idx].y
/*0080*/ @P0 IADD.X R7, R4, c [0x2] [0x1c];  // Compute high 32 bits of &b[idx]
/*0088*/ @P0 FFMA R2, R5, R2, c [0x2] [0x0]; // Compute v1 * a[idx].x + v0
/*0090*/ @P0 FFMA R0, R3, c [0x2] [0x8], R2; // Compute v2 * a[idx].y + above
/*0098*/ @P0 ST.E [R6], R0;                    // Store result.
/*00a0*/ EXIT;
```

(*a*) Briefly describe what each instruction does referring to the `dots` code. For example, for the instruction at 0018 one might say "Compute `idx` = `blockIdx.x` * `blockDim.x` + `threadIdx.x`".

Solution appears above.

(*b*) Show the equivalent GF3 machine code based on the description given in the class notes. Note that the GF3 has input values automatically delivered to registers and results automatically moved to the next stage. Be sure to take' advantage of GF3 vector operations (though they aren't a perfect match).

For the GF3 the values of `v0`, `v1`, and `v2` would be placed in a constant register. Recall that the GF3 registers are 4-element vectors, so the values would be placed in just one register. In the GF3 an element of `a` (both `x` and `y`) would be automatically delivered to an input register. The result, `b`, would have to be placed in an output register. The program would consist of just a single instruction, a `DOT3` (dot product). Register `o[0]` is one of 16 output registers, `v[0]` and `c[1]` are input and constant registers, respectively.

```
DOT3 o[0], v[0], c[1]
```

(*c*) Comment on the difference in code size and possible differences in execution time on systems with the same number of floating-point units.

Wow, the GF3 code is much smaller!!! That's due to two factors: having values delivered and taken away by hardware, and having vector operations.

Suppose that a DOT product uses four floating-point units. (Which is not that unfair because CUDA cores can do fused multiply adds.)

A modern GPU with four CUDA cores could execute the code at a rate of $\frac{21}{4} = 5.25$ cycles per element. A GF3-like design, assuming the rest of the hardware could keep up, would execute at the rate of one cycle per element, that's more than five times faster!