**LSU EE 4702-1** **Homework 4** Solution **Due: 20 November 2012**

*For the problems below refer to Chapter 2, Programming Model, and Chapter 4, Hardware Implementation, in the NVIDIA CUDA Compute Unified Device Architecture Programming Guide available locally via*
`http://www.ece.lsu.edu/gp/refs/CUDA_C_Programming_Guide.pdf`.

**Problem 1:** The kernel below is launched in a configuration of grid size of $(1024, 1, 1)$ and block size of $(256, 1, 1)$. (The components are $(x, y, z)$.)

```
__global__ void dots_iterate1() {
  int thread_count = /* Omitted so things aren't too obvious. */;
  int idx_start = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int idx = idx_start; idx < array_size; idx += thread_count )
    b[idx] = v0 + v1 * a[idx]; }
```

($a$) How many threads are there? How many blocks? How many warps?

The grid size is the same as the number of blocks $1024 \times 1 \times 1 = 1024$.

The number of threads is the product of the block size and grid size: $1024 \times 256 = 2^{10}2^8 = 2^{18} = 262144$.

The number of warps in a block is $\lceil B/32 \rceil$, where $B$ is the block size. The number total number of warps is the blocks per warp times the number of warps. For this example that's $2^{18}/2^5 = 2^{13} = 8192$.

($b$) Suppose the array size is $5 \times 10^5 = 500\,000_{10} = 7\,a120_{16}$ elements. How many threads will perform two iterations? One iteration? Zero iterations?

Two iterations: 237856, one iteration 262144. Zero: none.

($c$) A GPU has four multiprocessors. Explain why launch configurations of four and eight blocks are each better than a launch configuration of six blocks. Use the code above as an example.

Each block is assigned to one multiprocessor (in other words a single block can't run on two multiprocessors at the same time). If the number of blocks is not a multiple of the number of multiprocessors then some multiprocessors will be assigned more blocks than others. If each block takes the same amount of time to execute, an uneven distribution will result in execution time being longer than it has to be. The smaller the number of blocks the worse the problem is.

For example, suppose the kernel above was launched in two configurations, both having 256-thread blocks. One configuration has six blocks each taking 80 ms (or about 325 iterations per thread) or one with eight blocks each taking 60 ms (or about 244 iterations per thread). For the six-block launch two multiprocessors will be assigned two blocks and two will be assigned one block. The kernel will finish in $2 \times 80\,\text{ms} = 160\,\text{ms}$. In the eight-block launch each multiprocessor will be two blocks, the kernel will finish in $2 \times 60\,\text{ms} = 120\,\text{ms}$.

Note that the launch configuration given in this problem, 1024 blocks, is large enough so that load imbalance is not much of a problem with current GPU sizes, where the larger ones have about 16 multiprocessors.

**Problem 2:** Consider the kernel code below. It is launched with a block size of 512 threads and a grid size of 64 blocks. The array has $2^{20} = 1\,048\,576$ elements.

```
__global__ void dots_iterate15()  {
  int thread_count = /* Omitted so things aren't too obvious. */;
  int tid = threadIdx.x + blockIdx.x * blockDim.x;  // Not used in code.
  int idx_start = blockIdx.x + threadIdx.x * gridDim.x;

  for ( int idx = idx_start; idx < array_size; idx += thread_count )
    b[idx] = v0 + v1 * a[idx];
}
```

(*a*) The table below shows various information about selected threads in the launch described above. The first three columns should be self-explanatory. The three columns headed `idx` show the element number (value of `idx` in the code above) accessed by the respective thread in the first, second, and third iteration of the `for` loop. Each row has at least one column filled. Fill the remaining columns.

Solution appears below. Notice that consecutive thread ids do **not** access consecutive elements. In fact, the situation is far worse. Adjacent arrray elements are accessed by **different** blocks. For example, element 0 is accessed by block 0 and element 1 by block 1. These may be on different multiprocessors. Since the memory system brings in data in 128-byte units, much of that data will go unused.

| tid | blockIdx.x | threadIdx.x | -- idx --- First Iter | -- idx --- Second Iter | -- idx --- Third Iter |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 32768 | 65536 |
| 1 | 0 | 1 | 64 | 32832 | 65600 |
| 2 | 0 | 2 | 128 | 32896 | 65664 |
| | | | | | |
| 0 | 0 | 0 | 0 | 32768 | 65536 |
| 512 | 1 | 0 | 1 | 32769 | 65537 |
| 1024 | 2 | 0 | 2 | 32770 | 65538 |
| 513 | 1 | 1 | 65 | 32833 | 65601 |