This handout leads you through three brief exercises that demonstrate features of the CUDA architecture. They are designed for use in a Computer Organization course where the students have minimal background; some are just out of CS 2 in Java and have no prior experience with C. That said, I think they serve as simple illustrations of particular concepts for more advanced students (and you!).

Transfer time

The first exercise emphasizes the cost of transfering data between the main system (called the *host*) and GPU. This exercise uses the file `addVectors.cu`, which is a simple program that adds two vectors of integers. First, compile and run the program:

```
nvcc -o addVectors addVectors.cu
./addVectors
```

You should get a printout with a time, which is how long the program took to add two vectors (of length 1,000,000).

Now let's examine the code itself. Right near the top is the definition of the function `kernel`:

```
__global__ void kernel(int* res, int* a, int* b) {
  //function that runs on GPU to do the addition
  //sets res[i] = a[i] + b[i]; each thread is responsible for one value of i

  int thread_id = threadIdx.x + blockIdx.x*blockDim.x;
  if(thread_id < N) {
    res[thread_id] = a[thread_id] + b[thread_id];
  }
}
```

This is a pretty standard function to add the vectors `a` and `b`, storing the sum into vector `res`.

Let's see how this time breaks down between the data transfer between the main system and the GPU. Open the file and comment out the line that calls the kernel:

```
kernel<<<blocks,threads>>>(res_dev, a_dev, b_dev);
```

(This is the 3rd time "kernel" appears in the file and occurs near the middle of `main`.) Then recompile and run the program again. The program is now transferring the data back and forth, but not actually performing the addition. You'll see that the running time hasn't changed much. This program spends most of its time transferring data because the computation does very little to each piece of data and can do that part in parallel.

To see this another way, open the file again and uncomment the kernel call. Then comment out the lines that transfer the data to the GPU; these are in the the paragraph commented as "transfer a and b to the GPU". Then modify the kernel to use `thread_id` instead of `a[thread_id]` and `b[thread_id]`. (The program initializes `a[i]` and `b[i]` to both be `i`; see the "set up contents of a and b" paragraph.) The resulting program should be equivalent to the original one except that instead of having the CPU initialize the vectors and then copy them to the graphics card, the graphics card is using its knowledge of their value to compute the sum, thus avoiding the first data transfer. Recompile and rerun this program; now it is considerably faster. (We're no longer copying the two vectors, which are each a million entries long...)

This example shows the cost of transfering data, which can limit performance, particularly for an operation such as vector addition which is very computation light.

Thread divergence

Now we move on to our second factor affecting the performance of GPU programs. The factor is thread divergence, the phenomenon in which the program will run more slowly when the threads in a warp wish to

execute different instructions. Essentially, the system allows one cycle for each instruction that any thread in the warp wishes to run. All threads spend a cycle for each such instruction, with threads not wishing to execute that instruction running a nop instead.

To illustrate this property, we use the file `divergence.cu`. This file contains two kernels, creatively named `kernel_1` and `kernel_2`. Examine them and verify that they should produce the same result, which is to count the number of threads whose id takes on each remainder when divided by 32.

Compile and run this program:

```
nvcc -o divergence divergence.cu
./divergence
```

Then modify the code to use the other kernel, recompile, and rerun. You'll see that the running times are quite different.

That there is a difference is not terribly surprising since the kernels do use different code. To further explore this difference, change the number of cases enumerated in `kernel_2` (either delete some or use cut-and-paste to add more). You'll see that its running time changes, which should not happen; `switch` statements typically have running time independent of the number of cases.since they're just a table lookup.

This example emphasizes the SIMD nature of CUDA hardware and the danger of having control flow that varies between threads.

Constant memory

The third example shows one of the special kinds of memory in CUDA, constant memory. This memory is for values that will not be changed. Using it allows the hardware to cache values as a way of reducing memory traffic. It also uses another optimization for this goal: the result of reads to constant memory are broadcast to all threads in a half-warp, greatly speeding up the code if these threads all want the same value.

The code to illustrate this is in the `constMemory` and `constMemory-noX` directories. If you're on the servers we provided, use the `noX` version. If you're on your own system (and X works), use the other one so you get the graphical output. This code is taken from Chapter 6 of "CUDA by Example" by Sanders and Kandrot (modified only to compile with all the headers in the same directory and to remove the X calls for that version). The application creates an image of randomly placed spheres. Each pixel determines the closest pixel down the z axis from its location and sets its color accordingly.

There are two implementations of this in separate source files. Compile them both by typing `make`. The program `noconst` (made from `ray_noconst.cu`) is a "plain vanilla" implementation, while `const` (made from `ray.cu`) puts the sphere information into constant memory to exploit the fact that each thread examines them in the same order. The program generate the same output (up to differences in the random numbers used to place the spheres), but the constant memory version runs faster.

Examine the code to see the differences in declaring the variable `s` for the spheres and populating it.

Then change the `for` loop in the kernel so its first two lines are the following:

```
for(int j=0; j<SPHERES; j++) {
  int i = (j+threadIdx.x) % SPHERES;
```

This has the effect of making different threads in a half warp examine the spheres in a different order, slowing the program by making the broadcast of values from constant memory into a disadvantage since the requests are serialized. Clearly memory usage needs to be considered carefully in order to keep the many simultaneous threads running smoothly.

For questions after the conference: David Bunde (`dbunde@knox.edu`)