Lab 2

In this lab, you’ll compare the running times for CPU- and GPU-based implementations of the vector and/or matrix adding programs.

Timing

Our goal in using CUDA is to speed up our programs. The large number of cores on a GPU is good for this, but there is a factor that makes it harder: in order to use those cores, we need to move the data back and forth between the GPU. Today, we time CPU- and GPU-based implementations of the same task to understand the tradeoff between parallelism and data-transfer overhead.

Begin with either the vector-adding program I gave you or the matrix-adding program that you developed last time. Within the same program, write a function that performs the addition using the CPU (a for loop or two should do the trick). Once you’ve verified that your new version works, comment out the output parts of the program (you’ll be using big data sets and won’t want to see all that...).

Next, add infrastructure to time these two different implementations. Use the gettimeofday function to get the system time before the GPU-based calculation, between the calculations, and then again after the CPU-based calculation. (Use the man page for gettimeofday to learn about this function.) By subtracting these times, you should be able to getting the running times for each implementation. To make the comparison fair, be sure that you’re not including the time to initialize the arrays. For the GPU-based implementation, we want to compare the time to transfer the data, compute the result, and transfer it back. (Arguably, you should also include the time to allocate the memory on the GPU. I didn’t include this time since the same memory could be reused within a program, but you could count it since this allocation operation is required to use the GPU, but not in a purely CPU-based solution.) For the CPU-based implementation, just time your for loop(s). If you’re using the vector addition problem, you’ll also need to change the way that your arrays are allocated; use malloc instead of statically allocating them. (Statically allocated arrays live on the stack, which is exhausted for large problem sizes, whereas memory that is allocated dynamically with malloc is taken from the heap.)

Compare the two times as you vary the problem size N. What I’d expect is that the GPU-based version would start slower than the CPU-based version, but gradually overtake it as the problem size increases. This is not what I saw for the vector-addition problem; the CPU-based version remained faster even for the largest problem sizes I could run. (This is running on descartes.) The problem was that the (serial) data movement was too expensive compared to the (parallelizable) computation. To fix this, I changed the function being computed from addition to something that involved more work; my new function computed each c[i] from a[i] and b[i] using 6 multiplications and 3 additions. Introduce a more complicated function, being sure to modify the CPU- and GPU-based versions in the same way to keep the comparison fair. After this change, you should see the expected behavior: the GPU-based version starts slower for small problem sizes but gradually overtakes the CPU-based version as N increases.

Once you’ve been able to find a crossover point beyond which the GPU-based implementation outperforms the CPU-based one, vary what you’re doing and see how the crossover point changes. Does the crossover point depend on which machine you’re running on? (The cards have different numbers of cores and are connected differently.) How does it change with the complexity of the operation being computed? How does it change if we count the time to allocate memory on the GPU? Explore these questions for the rest of the period and see what conclusions you can draw. You should also coordinate with your classmates so that you’re not both running simultaneously on the same system, which will throw off your timing results.