Lab session 4:

- **Write a buggy kernel with a race condition: Each thread should try to add one to the same memory location. Launch this kernel with different block and thread numbers, and print out the final value of the memory location in each case.**
- **Write a better kernel which uses atomic addition for the same task, and print the new results - they should now be deterministic.**
- **Last week you used Thrust's builtin `plus` functor to reduce a vector. Now write a `coarsePlus` functor which adds together `n` elements of a vector, where `n` is specified in the functor constructor, and use this functor as the unary operator in a `transform_reduce` call that takes the sum of the elements in a vector. That is, we want to do a reduction in two steps: First the unary operator will add together N elements of the vector, then the binary operator will be the usual `thrust::plus`. Notice that in this case you do not want to work with a `thrust::device_vector` iterator, because then Thrust will assume that your operator only wants one element of the vector. Instead, use a `constant_iterator` whose value is a pointer to a device-side array, and have your operator method calculate the appropriate index range into that array. (You may find it convenient to use a `counting_iterator` to give the index of the thread.) This sort of mixture of raw pointers with Thrust iterators is a common pattern. Check your result, and the time it takes, against an ordinary**

reduction using the builtin `plus`; try with a thousand, a million, and a hundred million elements, and with `n=8, 25, 100`.

- **This exercise is a simplified version of the Coulomb-potential example. Consider a one-dimensional line along which are spaced, at regular intervals, 100 point charges. We want to calculate the potential due to these charges at every point. First, generate a vector of 100 random numbers; these are the sizes of the charges. We'll assume that the points are 1 unit apart. Make two kernels that calculate the potential at each point; one which does so by scattering (each thread reads one charge size and updates each point in the output array using `atomicAdd` (note that compute capability 2.0 introduced atomic operations for floats)); and one that does so by gathering, so that each thread writes to one element of the output array. Print the respective times taken, and check that the difference between corresponding elements of each vector is less than some reasonable tolerance, perhaps 0.01. (Because floating-point addition doesn't commute and has roundoff errors, the numbers won't be completely identical.) You may do this exercise either by Thrust methods or by unadorned CUDA coding.**