

Practical 4: reduction operation

The main objectives in this practical are to learn about:

- how to use dynamically-sized **shared** memory
- the importance of thread synchronisation
- how to implement global reduction, a key requirement for many application codes
- how to use shuffle instructions

What you are to do is as follows:

1. Read through the **reduction.cu** source file and note the following:
 - The main code computes the results using both the CPU and the GPU. The CPU code is very simple, whereas the GPU code is much more complex.
 - Try to understand the **reduction** kernel completely.
 - The kernel uses dynamically allocated shared memory; the size is a third argument in the <<< >>> brackets.
2. Use Nsight or the Makefile to generate the executable **reduction**. Run the program, and check that it gets the correct result.
3. As given, the code assumes the number of threads is a power of 2. How would you modify it to handle cases in which it is not a power of 2?
4. As given, the code performs the reduction operation for a single thread block. Modify the code to perform reduction using multiple blocks with each block working with a different section of the input array.

As explained in lectures, there are two ways in which the partial sums from each block can be summed:

- each block puts its partial sum into a different element of the output array, and then these are transferred to the host and summed there;
- an atomic addition or lock is used to safely increment a single global sum.

Try at least one of these.

5. If there is time, modify the block-level reduction to use shuffle instructions.
6. If there is time, modify the `laplace3d` example from Practical 3 to compute the root-mean-square change at each timestep. This will require a global reduction to sum the squared changes.