

GPUs: Reduction

1 Dot product

One task that is often required in scientific computing is that of reduction: applying an operation to an entire vector of values to reduce them to a single value. This includes operations such as finding the maximum of a vector or a sum of all vector elements.

1.1 Outline

Start with the code framework `DotProduct.cu`. This should calculate the dot product of two large vectors x and y . You should use CUDA to perform the dot product. The framework will allocate the data and check the result afterwards. There is an incorrect example already implemented as a demonstration.

There are various ways to do this, however, they all reduce (pun intended) to the same thing: use a single block to perform the sum for a subset of threads, and output to a smaller array. This array can then be fed into the same kernel, until the number of elements is sufficiently small. At this point, copy the array back to the host and perform a final sum there if necessary.

1.2 Atomic operations

First, try using an atomic function to do the addition: `atomicAdd(float* sum, float term);` on each thread.

1.3 Tree-based reduction

Either follow the approach presented in lectures, or the CUDA white-paper in `CUDA_SDK/6_Advanced/reduction/doc/reduction.pdf`. You should first do a reduction across a single block only, reducing the number of elements needing to be summed by a factor of 512. This result can then be copied back to the CPU where the sum can be reduced further.

1.4 Multiple tree-based reduction

Instead of copying the data back to the host after it has been reduced once, write another kernel to do the reduction on the GPU. This can be called repeatedly until the sum has been reduced to a single number that can be copied back to the CPU.

1.5 Points to consider

Recall that floating-point arithmetic is not associative and accuracy can be compromised if values cover a wide range of orders. How might this be avoided? If you find an approach that appears suitable for ordinary serial operations, remember that you may be summing in a different order on the GPU to that on the CPU.

1.6 The even easier way...

Try using the Thrust library to calculate the dot product instead using `thrust::inner_product`. This will require no kernel writing on your part. Compare the performance to the kernel(s) you have written.

1.7 Solution

A solution is found in `solutions/DotProduct.cu` comparing various different methods. Example timings on a Kepler K20 are:

CPU time = 60 ms

Naive approach - wrong:

Multiplication wrong! $9.39725 \neq -145.589$

Time: 32.3672ms

Using atomic operations:

Multiplication correct! $-145.599 = -145.589$

Time: 86.0549ms

Reduction across one thread block only:

Multiplication correct! $-145.592 = -145.589$

Time: 5.47562ms

Repeated reduction:

Multiplication correct! $-145.592 = -145.589$

Time: 5.51338ms

Thrust:

Multiplication correct! $-145.591 = -145.589$

Time: 2.3097ms