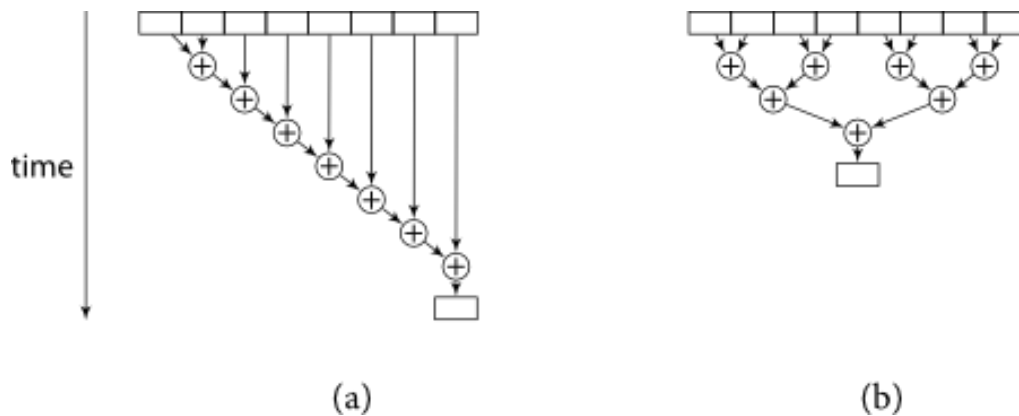


Introduction

We have already learned how to access/use GPU device properties. We started looking into parallel reduction sum implemented on CUDA. The picture below shows the fundamental difference between standard (a) and parallel reduction sum (b).



Our first implementation of this algorithm was already performing quite well compared with CPU, however, it was far from optimal. Our goal this week is to look at different ways of improving the performance of `reduceKernel1`. The main objective here is to get a basic idea about different ways and levels of performance optimisation on CUDA.

There are two ways of measuring the performance of CUDA kernels depending on whether a particular kernel is compute or memory bound. We would use GFLOPS/sec for compute-bound kernels (high arithmetic intensity) and bandwidth for memory-bound kernels. Parallel reduction sum has a very low arithmetic intensity (1 flop per loaded element) so our goal will be to maximise the bandwidth. For example GTX680 has memory bandwidth of 183.35GB/sec, however, the very first implementation of our kernel is far from that value.

Try to download the `reduce_prof_v1.cu` from the portal and run it and you should see the effective bandwidth, which is calculated by timing specific program activities and by knowing how data is accessed by the program.

$$\text{Effective bandwidth} = ((Br + Bw) / 10^9) / \text{time}$$

Here, the effective bandwidth is in units of GB/s, Br is the number of bytes read per kernel, Bw is the number of bytes written per kernel, and time is given in seconds.

For example, to compute the effective bandwidth of a 2048 x 2048 matrix copy, the following formula could be used:

CUDA labs

$$\text{Effective bandwidth} = ((2048^2 \times 4 \times 2) / 10^9) / \text{time}$$

The number of elements is multiplied by the size of each element (4 bytes for a float), multiplied by 2 (because of the read and write), divided by 10^9 (or 1024^3) to obtain GB of memory transferred. This number is divided by the time in seconds to obtain GB/s.

The *V1 kernel* uses very slow mod operator, which hits the performance. In addition, there are lots of shared memory bank conflicts (multiple threads accessing same data), lots of idle threads and instructions overheads. Follow [this presentation](#) and try to implement the first five kernels. For additional explanations, see [this presentation](#) focused on optimisations.

All these implementations are also uploaded on the portal so if you struggle try to have a look there. Below are the results from GTX680

V1 - time: 1.048288 ms throughput: 15.2630 GB/s

V2 - time: 0.731552 ms throughput: 21.8713 GB/s

V3 - time: 0.576064 ms throughput: 27.7747 GB/s

V4 - time: 0.338624 ms throughput: 47.2500 GB/s

V5 - time: 0.296864 ms throughput: 53.8967 GB/s

Additional optimisations using complete unrolling and multiple sums per thread:

V6 - time: 0.236672 ms throughput: 67.6041 GB/s

V7 - time: 0.118112 ms throughput: 135.4646 GB/s

Once you are done you can try to get familiar with NVIDA Visual Profiler where you can load and profile each of these implementations and see how they perform. For a getting started guide see the Visual Profiler section [here](#).

Additional Exercise

Download the source code, which is next to the link where you downloaded this document. Open the `gmem_reverse.cu`, which reverses an array using global memory. Try to go through the code and understand it and once you do then try to think about how you could optimize it. We will be around to help you and if you get stuck you can always have a sneak peek into the `smem_reverse.cu`, which is also attached.

CUDA labs