Reduction data access pattern – Accelerating C++ with CUDA advanced class.

14.12.2020 Introduction to CUDA and OpenCL

Task:

The purpose of this lab is to implement an algorithm that performs reduction of a one-dimensional input array. The final output should be the sum of the all elements in the array. The program should handle a list of an arbitrary length. For our lab we restrict the input to array of size ~$\mathcal{O}(10^8)$. Pay special attention to the "boundary conditions", where the length of the vector is not a multiple of the created processing block size. The partial sum, crated by respective blocks can be either handled by the GPU or send back to the host to be finished by CPU.

Hints:

To increase the efficiency of the calculations – make all the threads in block work in first iteration and define the input array segment as twice as long as the number of threads in the processing block.

Must be:

In the final solution, we should have in the code:

- A function that checks and handles error queries to the GPU
- Processing grid optimisation taking into account the input data vector size, the amount of shared memory available per SM/processing block
- A wrapper function that handles the kernel launch with appropriate arguments allowing the processing grid to be worked out

When creating the report please include the following answers

- Why the reduction algorithm is so vital, start from naming a few problems that represent this type of processing?
- Analyse the workload of each thread. Estimate the min, max and average number of real operations that a thread will perform (i.e., when a thread calculate a number contributing to the final result). How many times a single block will synchronise to before finishing its calculations (until a single partial sum is obtained)?
- Describe the possible optimisations to the code?
- Is the use of atomic operations advisable in this case?
- Think in a more abstract way about the reduction algorithm. Does the operation being parallelised need to be commutative?

General template:

1. Necessary includes
2. Define the max bin constant
3. Define the error checking/handling function
4. Define the reduction wrapper function
5. Define the main function able to read line arguments for the data size

6. Create single threaded application for the reduction

Structure of the main function:

1. Read the input data size
2. Define the data structures using malloc low level functions
3. Allocate necessary memory on host and device
4. Generate data set (for this exercise do not initialise the data on the device!)
5. Create timer object (as you learned from the matrix multiplication algorithm)
6. Transfer the data
7. Run the reduction wrapper
8. Fetch the data back form the GPU and finish the computations
9. Compare the output with the single threaded application