GPU Programming 2019/20

Optimization of Reduction

- Code is written for arrays of size 2^k (for sufficiently large k; one multiprocessor has to be completly filled which is 2 × `blockSize` starting in `reduction4.cu` and 4 × `blockSize` starting in `reduction6.cu`).
- Inspired by Mark Harris' optimization of reduction.[1]

Versions

`reduction_cpu.cpp`

- Simple C/C++ implementation.

`reduction0.cu`

- Naive[2] implementation with one thread.

`reduction1.cu`

- Naive implementation based on programming model with global memory used directly.
- No synchronization between blocks → race condition.

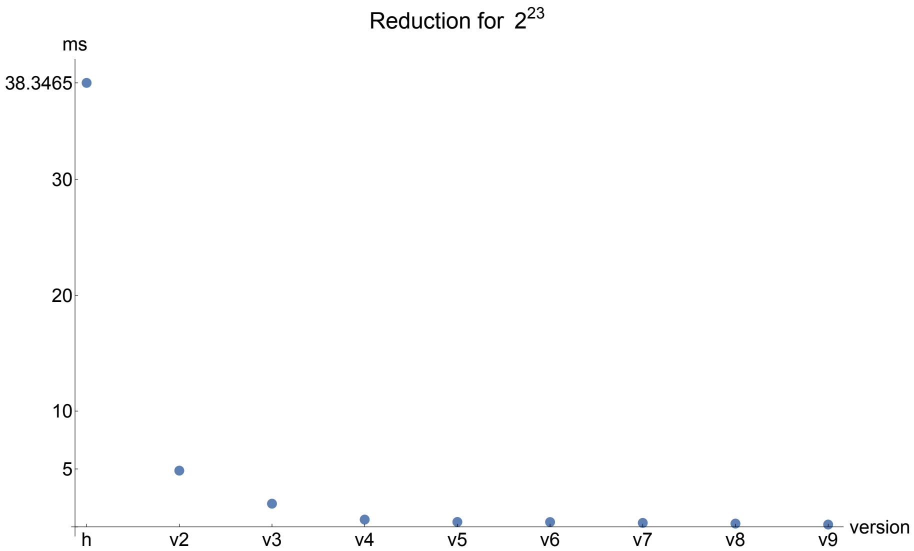
`reduction2.cu`

- Global synchronization between each tree levels on CPU (very high overhead).
- Coalescing in global memory access reduces with each tree level.

`reduction3.cu`

- Shared memory to reduce global memory access and avoid penalty of non-coalesced access.
- Final reduction beyond blocks on CPU.

[1] `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`



`reduction4.cu`

- Double amount of work per block.
- Avoid divergent branches by using index instead of `threadIx.x` to index data.

`reduction5.cu`

- Avoid bank conflicts by reversing loop from large to small stride. Shared memory access is now by threadIdx and threadIdx + const.

`reduction6.cu`

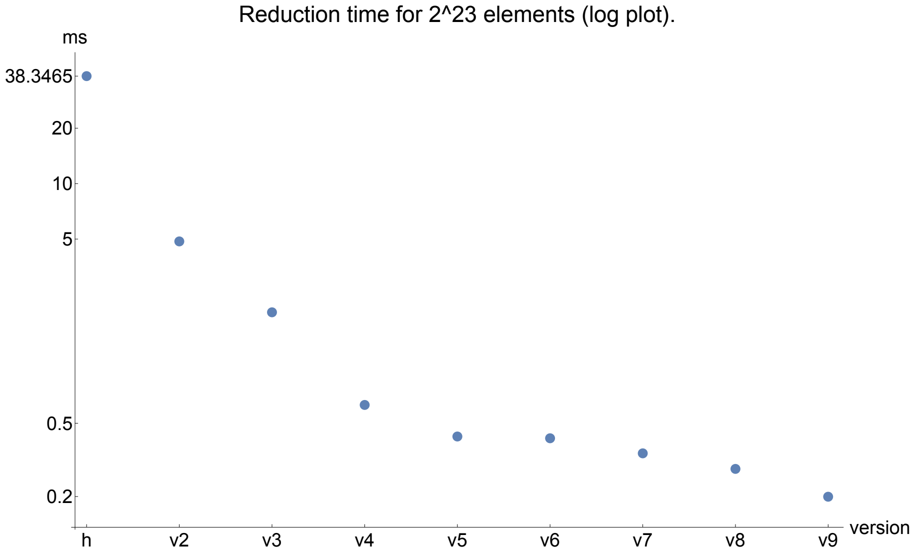
- The reduction at the finest level can be performed in the load step.

`reduction7.cu`

- Loop unrolling for last levels.

`reduction8.cu`

- All loops fully unrolled. Templated to remain flexible with respect to thread block size.



`reduction9.cu`

- Process multiple elements per thread (motivated by theoretical analysis).
- Multiple elements are processed directly from global memory.