GPU Accelerated Lattice Boltzmann Solver

Fredrik Robertsen

Åbo Akademi

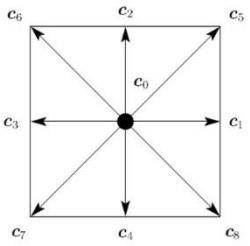
froberts@abo.fi

Outline

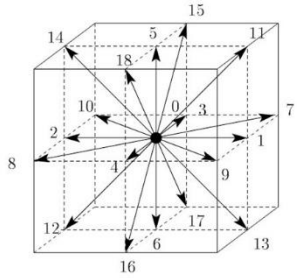
- Lattice Boltzmann intro
- General development
- Initial GPU porting
- Adaptation into a multi node solver
- Communication and computation overlap
- Performance results

Lattice Boltzmann intro (by an M.Sc.)

- Fluid dynamics solver
- Discretized fluid space
- Discretized velocity space

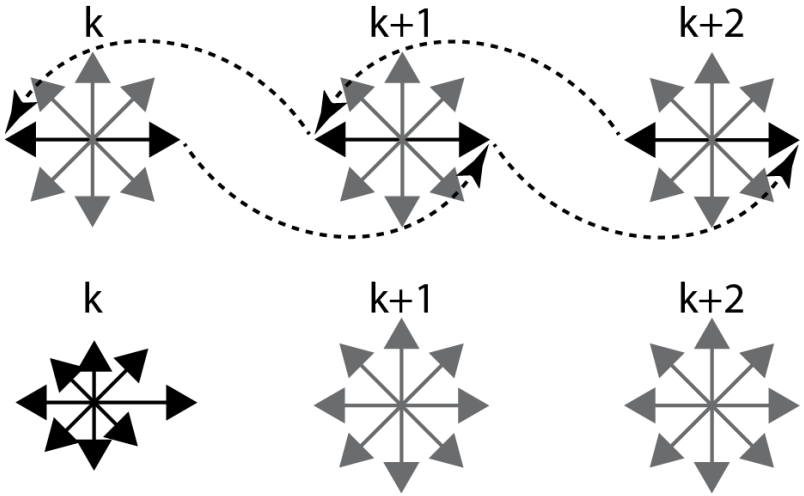


D2Q9



D3Q19

- Move data around ***(propagation)*** and mangle ***(relaxation)*** it



Hardware and software

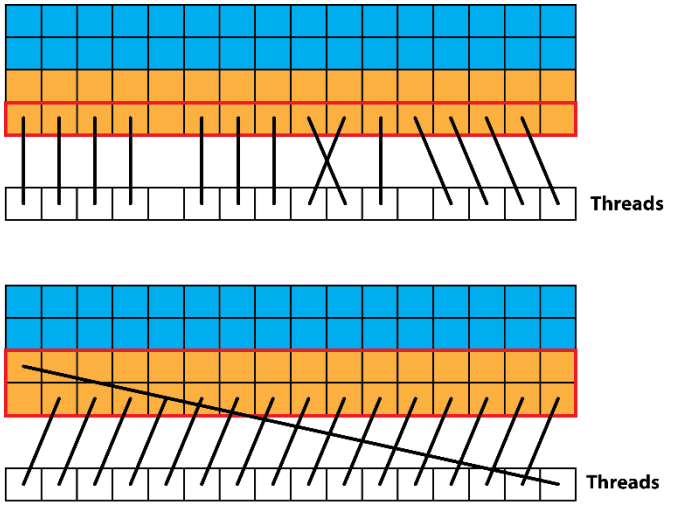
- Hardware used:
  - CSC:s hybrid prototype cluster
    - Nvidia K20 GPUs & K20x
    - FDR infiniband
  - Our own Nvidia K20
  - Our own cluster
    - 8 nodes with slightly older hardware
- Software
  - CUDA 4.2-5.5
  - GCC, newest version supported by CUDA at the time

Timeline

- 6 months from reference CPU solver to well performing single GPU
  - 90% of the code changed in the final version
- Adding multi GPU functionality took another 3 months
  - Original code was single thread and single compute node
- Including various tweaks, special versions and our Intel Xeon Phi work, I have been working on this for about 1.5 years.

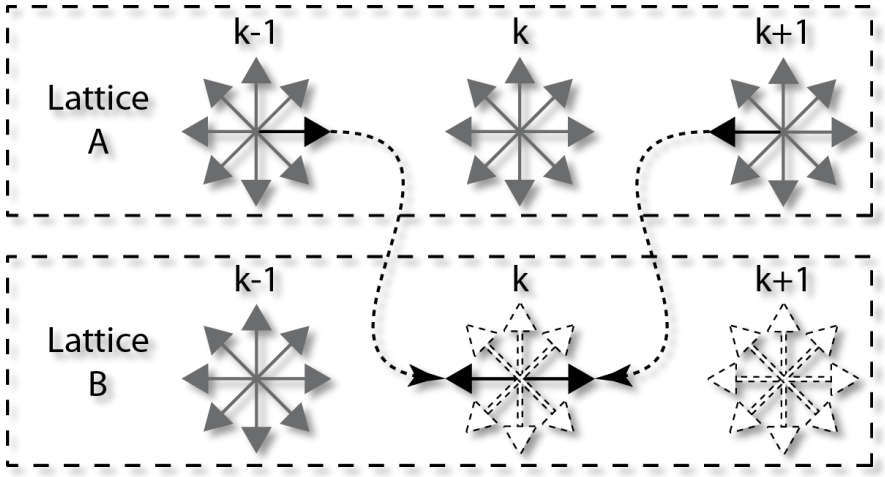
CUDA execution model

- Single instruction multiple threads
- Threads are executed in blocks
  - Divided into 32 thread warps
- Threads in blocks share a register file
- 128 byte aligned memory loads
  - 16 values for double
  - 32 values for float



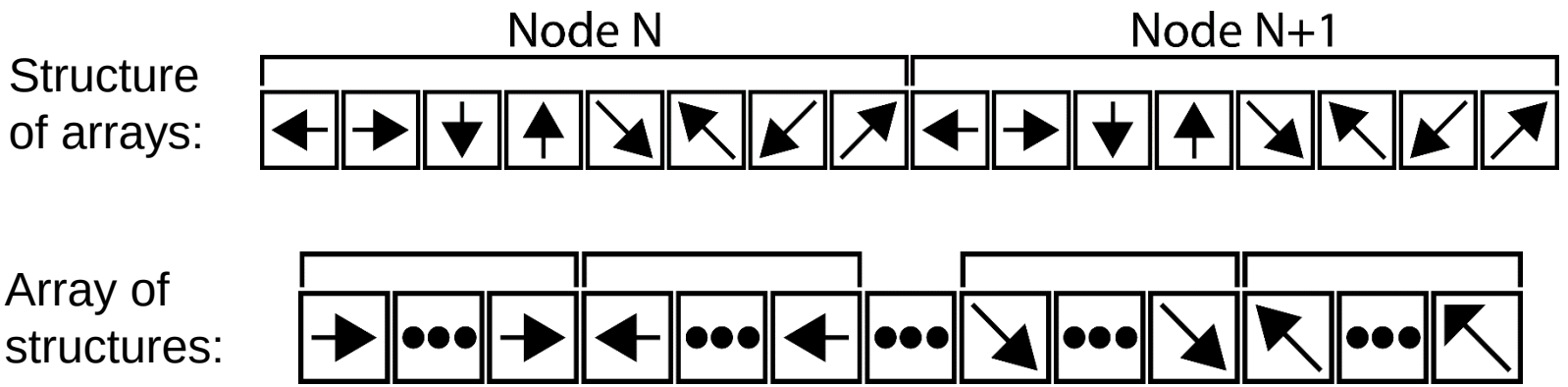
GPU adaptation

- Remove dependencies between threads
  - GPUs execute threads in no particular order
  - Two memory locations for each velocity vector
  - Read from one and write to the other
  - Swap pointers, repeat



Data layout

- In order to utilize the memory bandwidth available the data needs to be structured in a specific way
  - Threads are executed in blocks
  - Data is read in segments
  - As many threads in a block as possible should be “fed” from one data segment
  - Structure of arrays, NOT array of structures



Propagation step

- Reads can be orchestrated to be perfectly coalesced
- Writes cannot
  - Any real system we will simulate will include solid nodes
  - These solid nodes will cause some divergent writes within a warp
  - There is nothing that can be done about this without a radical change in algorithm
- Alternatively it can be the other way around, perfect writes and imperfect reads

Relaxation step

- Maps perfectly to the GPU architecture
- Streaming data
  - Data in → modify it → data out
- Reducing register usage
  - More threads running
- Spreading out the reads as much as possible
  - Avoiding having the kernel memory bound in one place and compute bound in another

Fusing relaxation and propagation

- Doing relaxation and propagation in different kernels require access to all data twice
- Either read the data from neighbors before relaxation or write it after relaxation
- ½ memory bandwidth needed:

"2X performance !!!!"

- In reality: 1,36 speedup

Getting rid of “useless” data

- Having data that is not used in the relaxation mixed in with the fluid data will cause coalescing issues
  - No data allocated for walls
  - No halo nodes
    - Bounce back at every border
    - Handle propagation trough them with the communication

Multi GPU

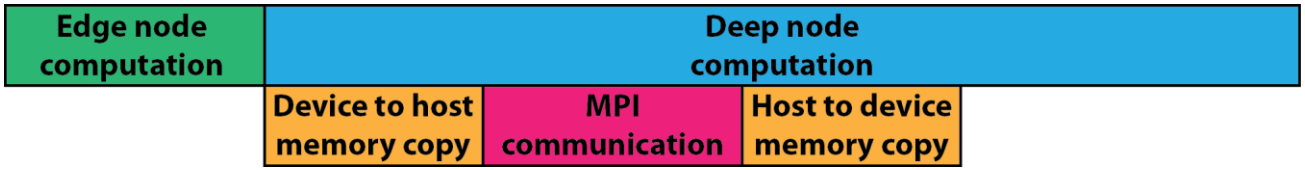
- Either speed up computation by distributing the problem, or run larger problems
- Introduces communication between compute nodes
- Values propagated outside the local domain needs to be transferred to neighbor nodes
- MPI for communication
  - Currently GPU→host→MPI→host→GPU
- Various MPI_Gather calls to reduce the data to a few values

Getting the data off the GPU

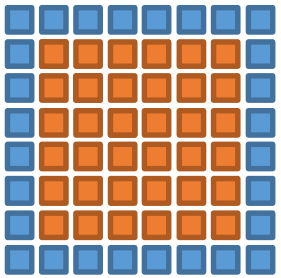
- Fastest way is to move all data in one transfers
- Edge data is scattered so it needs to be gathered up into continuous memory space
- And it needs to be scattered back out after communication
- Small somewhat inefficient kernel to gather the data
  - Scattered reads or writes
  - Low register usage → good occupancy

Overlapping computation and communication

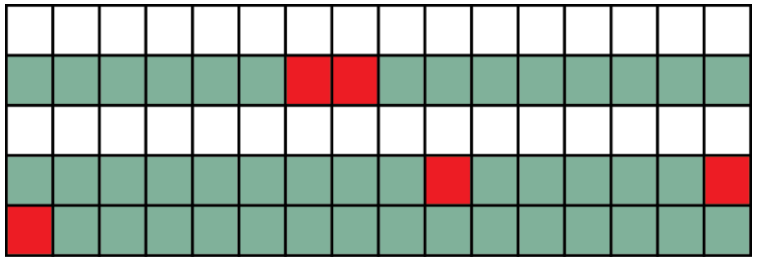
- The GPU can do multiple things at once
  - Computation
  - Memory transfers to and from the host
- Continue with computation as edge values are communicated



Dividing the computational domain



- Dividing the computational domain is not as trivial as it might seem
- The normal domain decomposition will lead to unaligned memory reads for both edge and deep kernels
- Instead divide using aligned segments around edge nodes
- Reduces computation that can be overlapped, but the better aligned access improves performance significantly

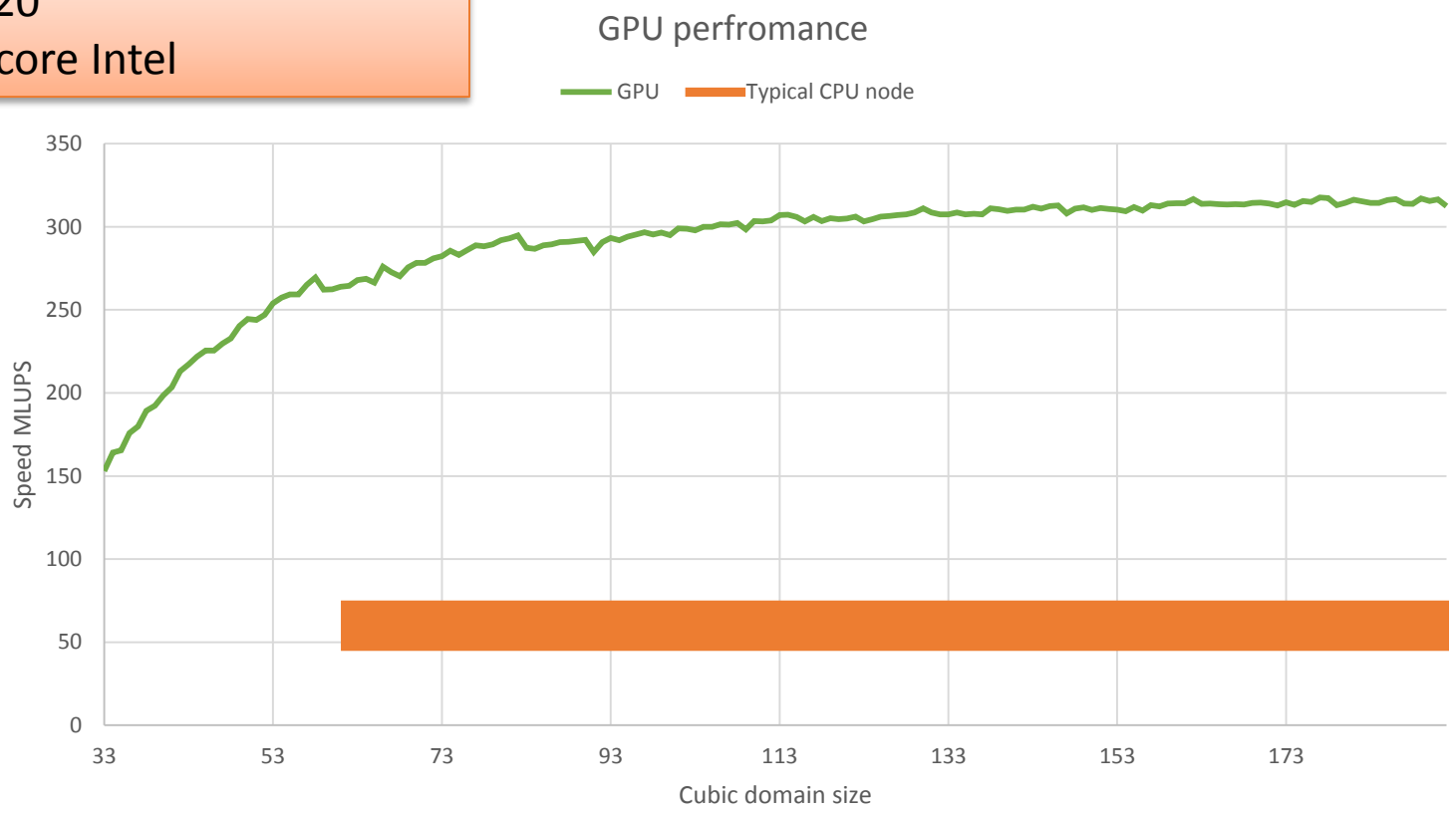


More asynchronous activities

- We want to know how the total macroscopic velocity and mass of the system changes over time
- We calculate the macroscopic values for each node in parallel on the GPU
- Move this to the host and reducing it to one value there
  - The host only does communication, leaving a lot of empty threads to calculate our statistic
  - Memory transfer can be done asynchronously
  - Segmented memory transfer so that it does not interrupt the communication

GPU results

GPU: NVIDA K20
CPU node: 16 core Intel



Thank you

Supported by the Cresta project

www.cresta-project.eu

Questions?

froberts@abo.fi