JÜLICH
FORSCHUNGSZENTRUM

S0036 - Multiparticle Collision Dynamics on one or more GPUs

May 15th, 2012 | Elmar Westphal - PGI/JCNS-TA Scientific IT-Systems

Multiparticle Collision Dynamics on one or more GPUs

- The MPC-algorithm:
  - Overview
  - Different steps
- The GPU implementation
- Benchmark results
- The multi-GPU implementation
- Outlook

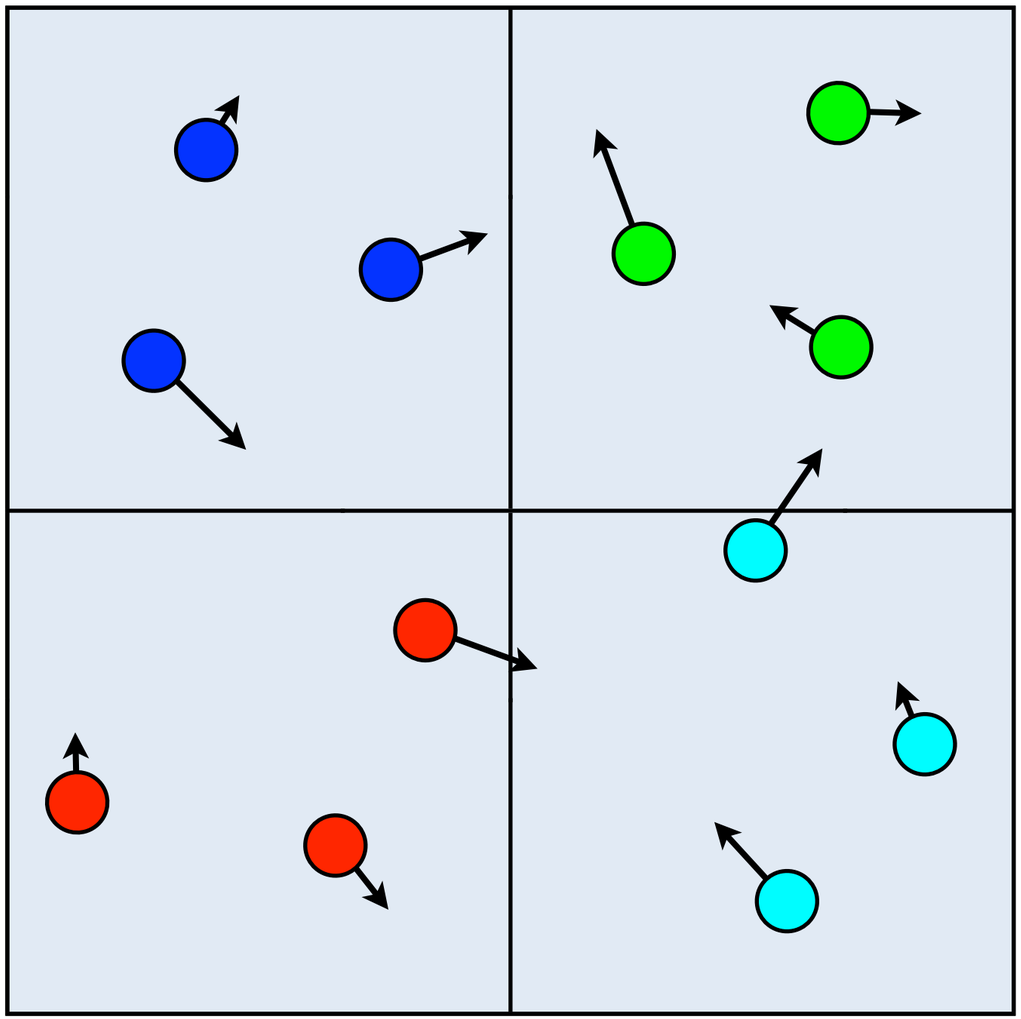
Multiparticle Collision Dynamics, Overview

- Mesoscale simulation method for hydrodynamic interaction
- Particle based
- Coarse grained
- Preserves system's kinetic energy and momentum
- Used in conjunction with conventional MD simulations

Multiparticle Collision Dynamics, Overview

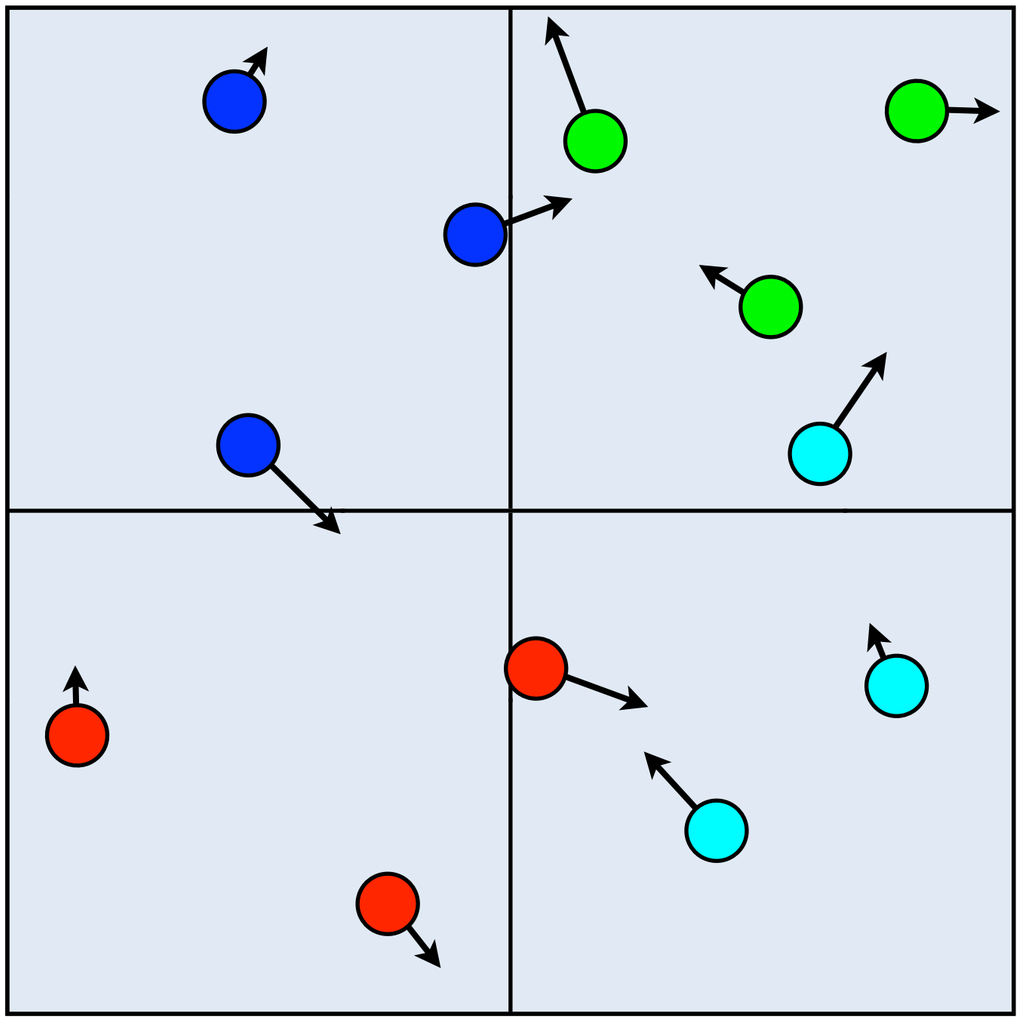
- The simulation system:
  - Divided into cubic boxes (cells) of equal size
  - Contains fluid and solute particles
  - Periodic boundary conditions apply
- The algorithm:
  - Integrate fluid particles
  - Shift lattice by random value
  - Rotate particles velocities of each cell around a random axis

Integration step



Note: the actual implementation is three-dimensional, but all slides show 2D schematics.

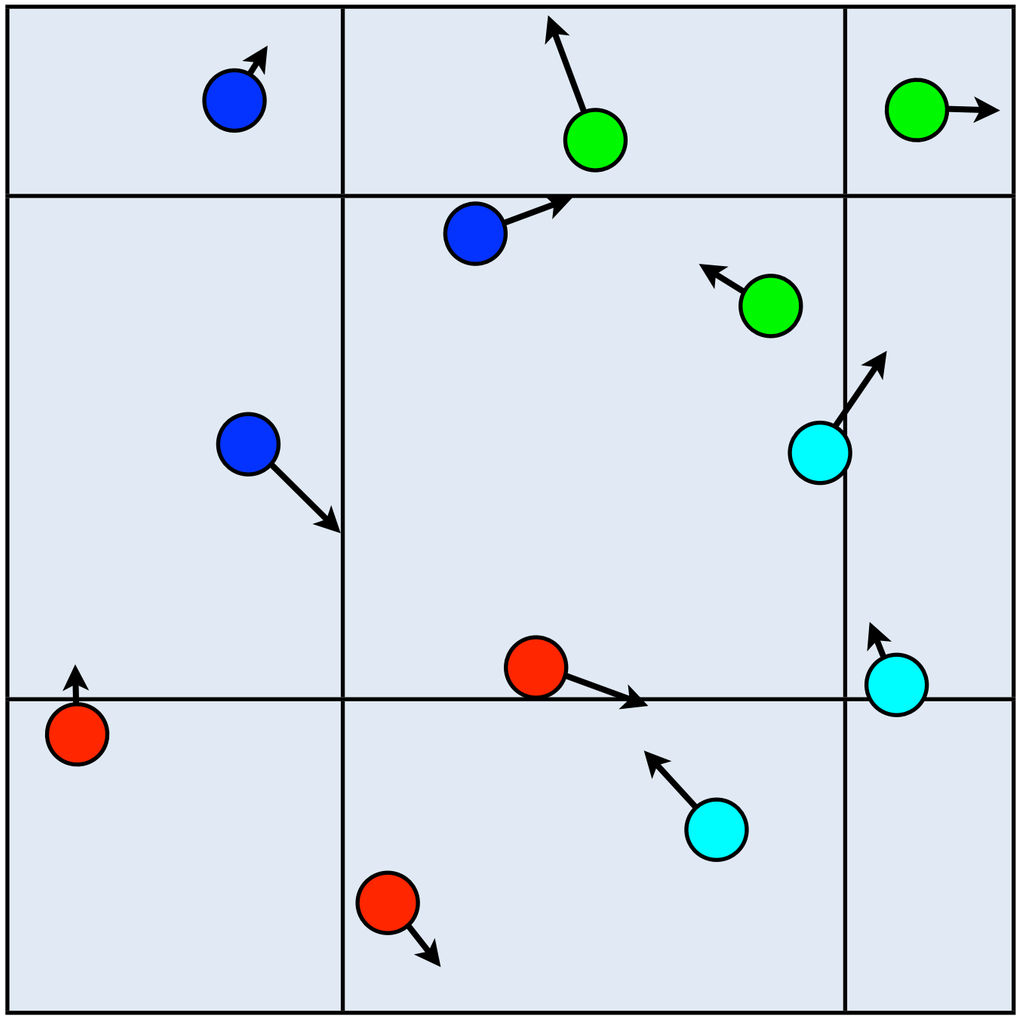
Integration step



- Move particles balistically

Note: the actual implementation is three-dimensional, but all slides show 2D schematics.

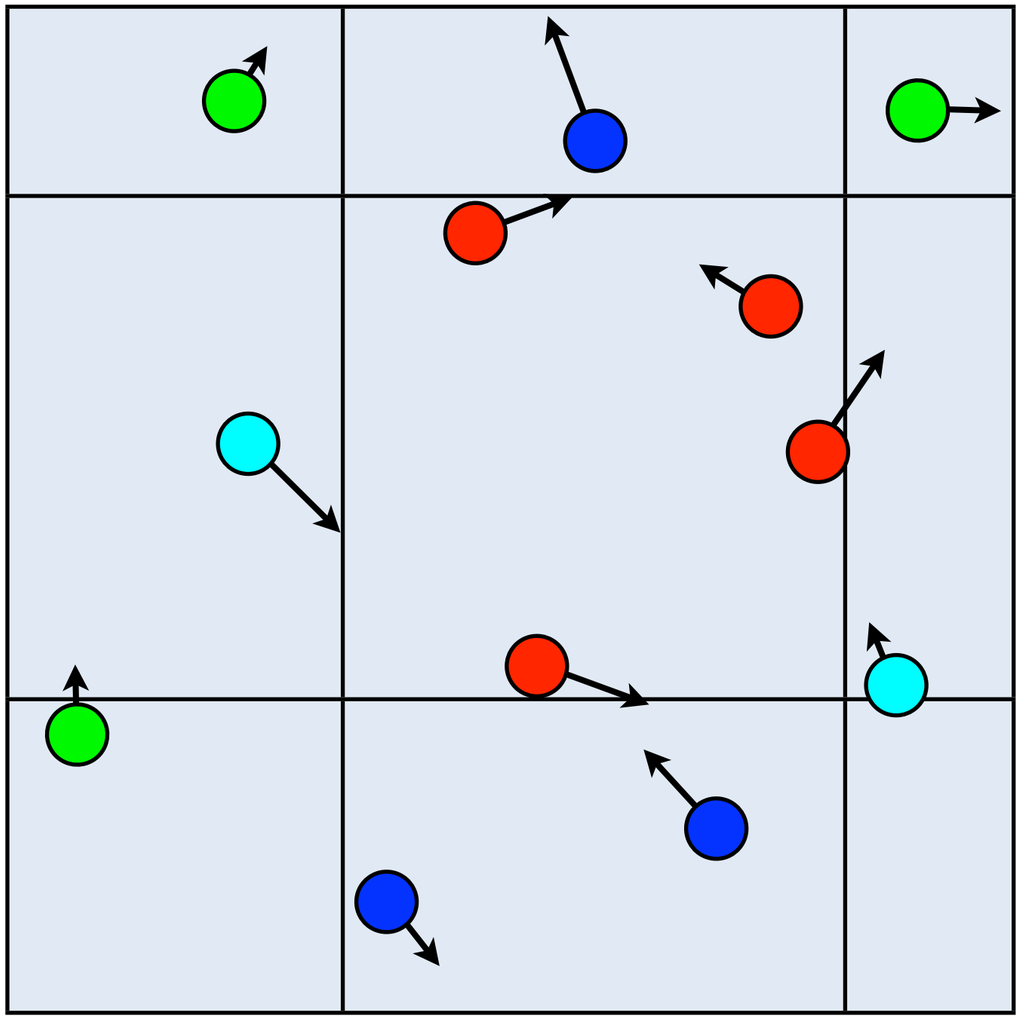
Integration step / Shift of lattice



- Move particles balistically
- Shift lattice by random value in each direction

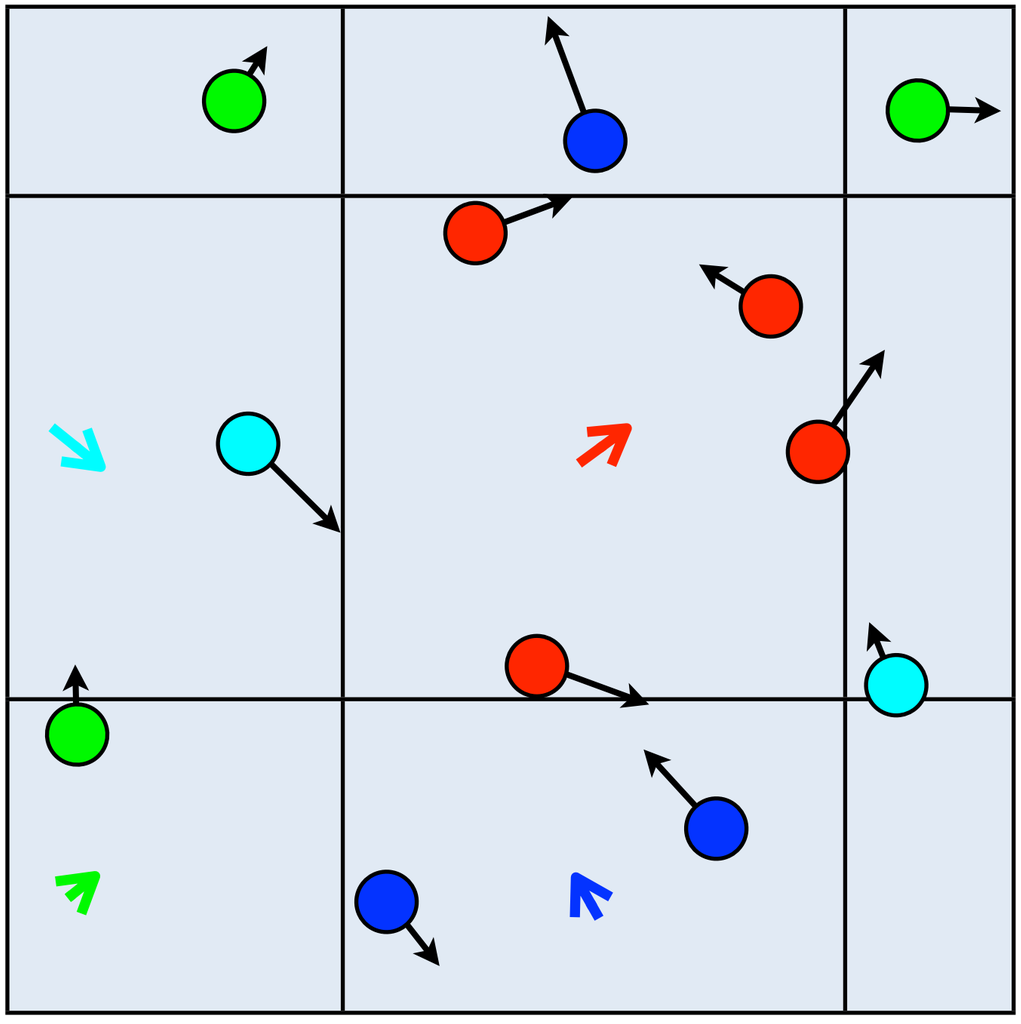
Note: the actual implementation is three-dimensional, but all slides show 2D schematics.

Collision step: Assignment to cells



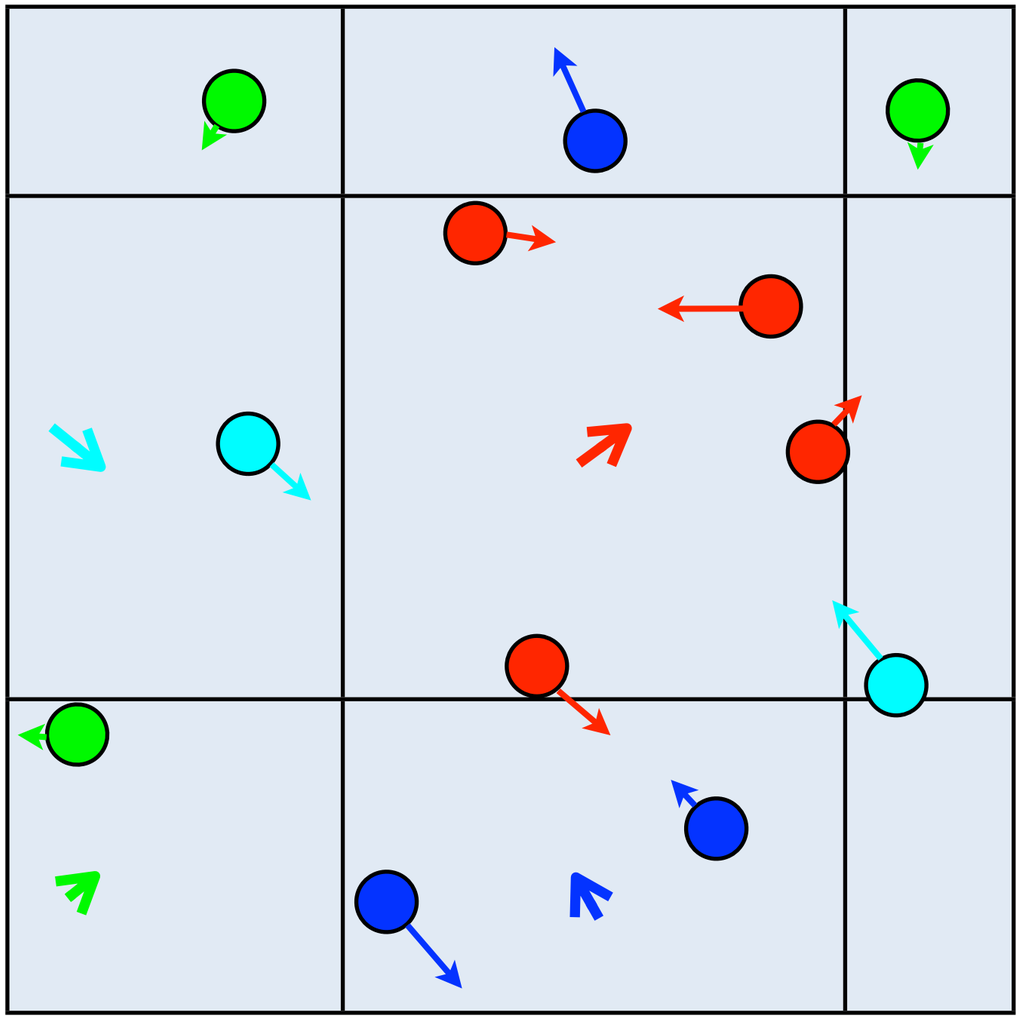
- Assign particles to cells according to shifted lattice

Collision step: Center of mass velocity



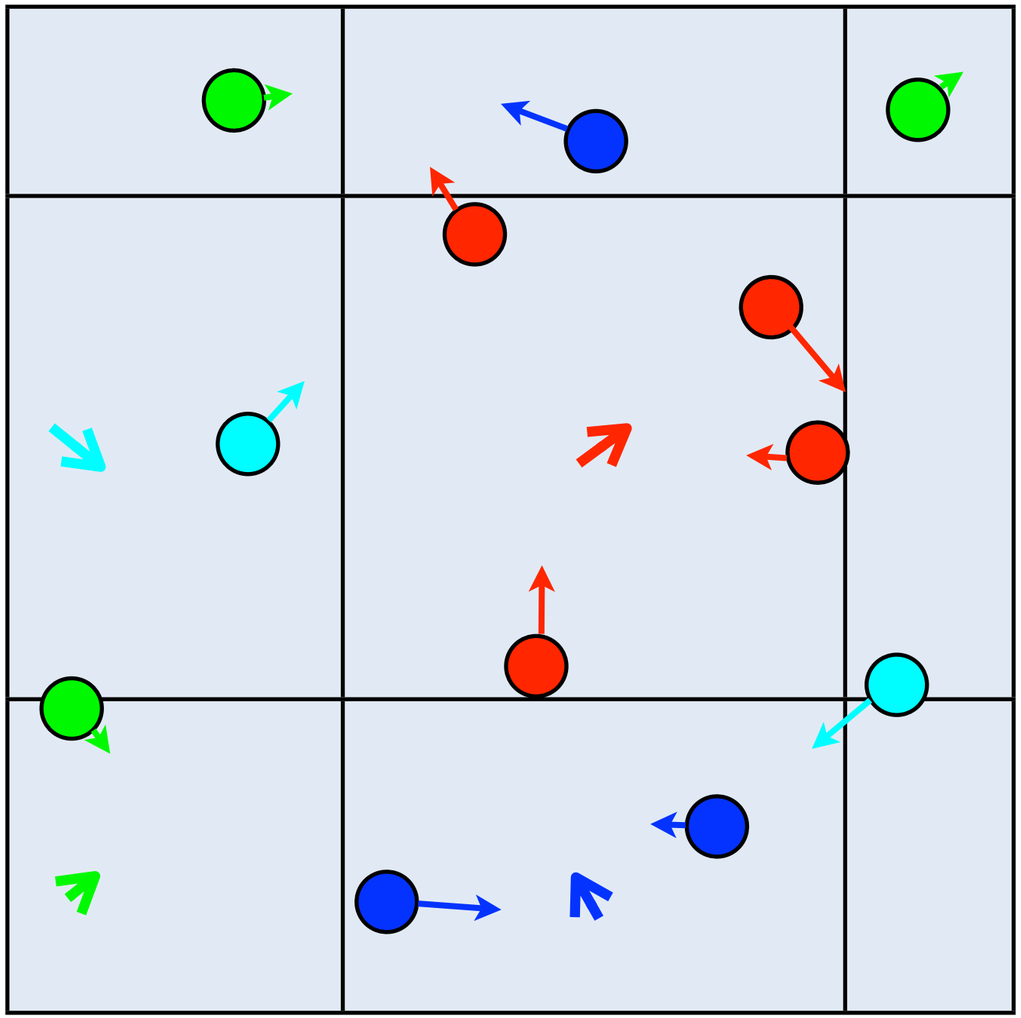
- Assign particles to cells according to shifted lattice
- Calculate center of mass velocity (v_{cm}) for each cell
  - v_{cm} is the sum of the cell's particle's kinetic momenta divided by their total mass

Rotation of relative velocities



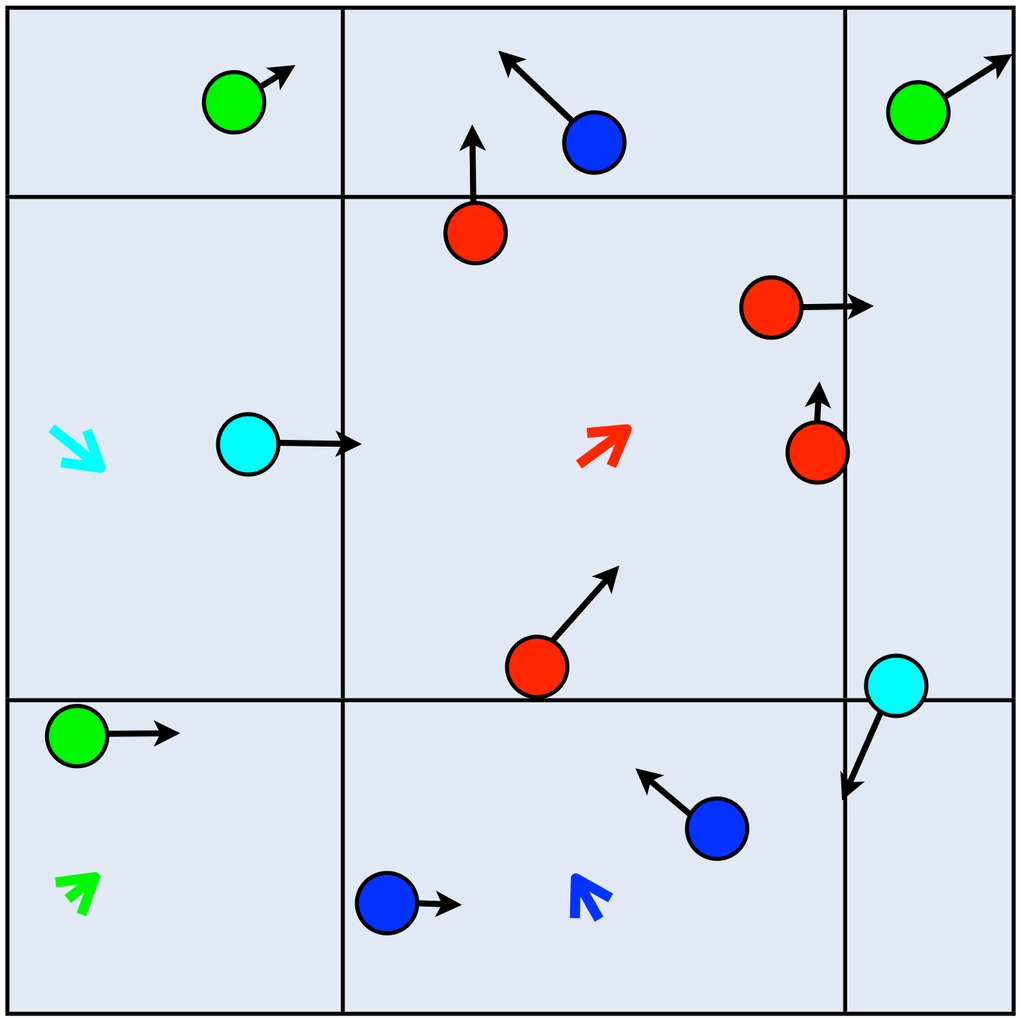
- Calculate relative particle velocity by subtracting V_{cm}

Rotation of relative velocities



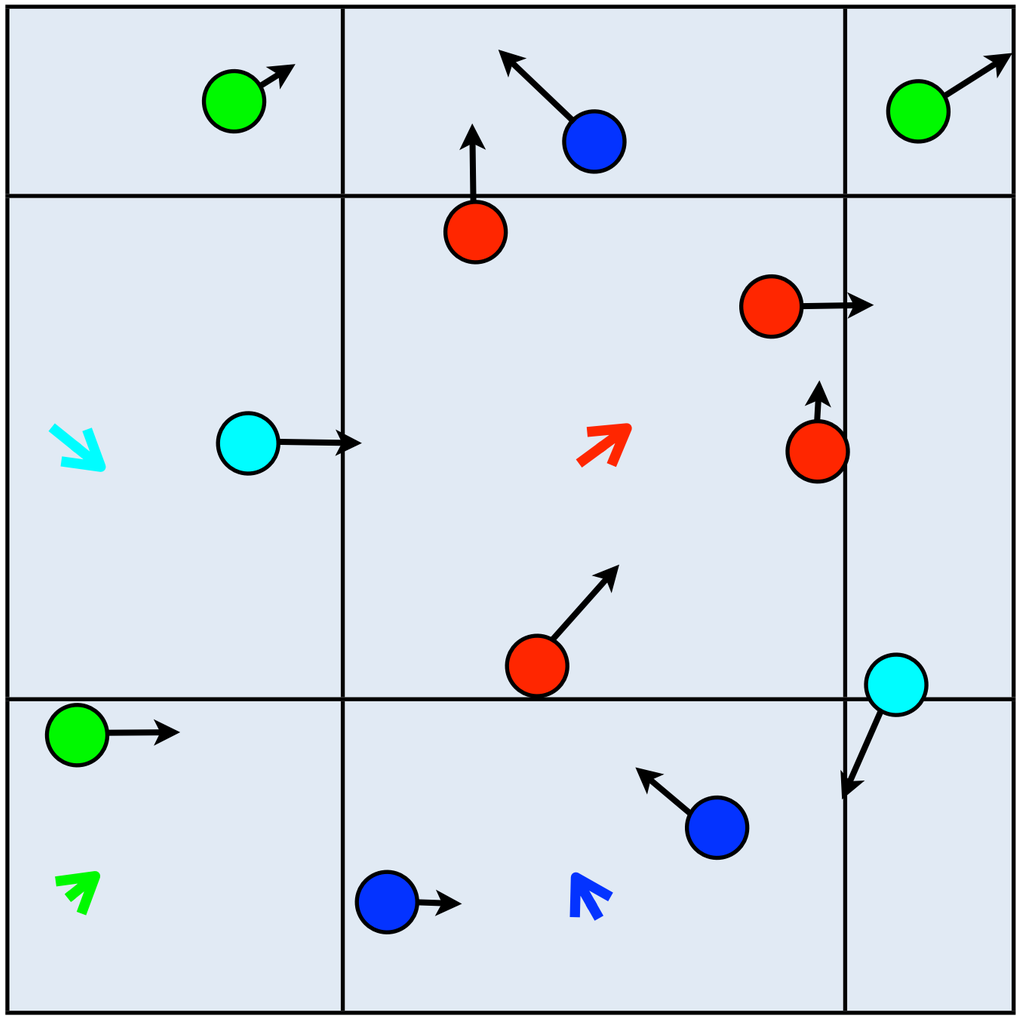
- Calculate relative particle velocity by subtracting V_{cm}
- Rotate relative velocity around random axis

Rotation of relative velocities



- Calculate relative particle velocity by subtracting V_{cm}
- Rotate relative velocity around random axis
- Add V_{cm} again

Rotation of relative velocities



- Calculate relative particle velocity by subtracting V_{cm}
- Rotate relative velocity around random axis
- Add V_{cm} again
- Perform MD and repeat

Multiparticle Collision Dynamics on one or more GPUs

- The GPU implementation:
  - Distribution of tasks
  - Streaming step (integration)
  - Different ways of calculating v_{cm}
  - The rotation step
  - Optimizations
  - Additional functionality
  - Achievable system sizes

Distribution of tasks and importance of individual particles

- MD simulations of solute particles are performed on CPU(s) or in additional GPU code
  - Individual solute particles matter at application level
- Hydrodynamic interactions between fluid and solute particles are performed on the GPU using MPC
  - Individual fluid particles do not matter at application level
- Communicating fluid particle properties is (too) expensive

Black Box concept for the fluid

- The fluid is considered a Black Box, its individual particles and the actual MPC step are invisible to the application:
  - transfer solute particle positions and velocities to GPU
  - perform MPC step
  - transfer changed solute velocities back
- Statistical properties of the fluid (kinetic energy, momentum) can be computed on the GPU

Implementation of the streaming step

- perfectly suitable for GPUs:
  - –all particles are processed independently
  - –all data can be accessed linearly (coalesced if stored correctly)
  - –streaming step is memory bound
- Procedure:
  - –read particle position and velocity
  - –propagate particles and apply periodic boundary conditions
  - –write new position

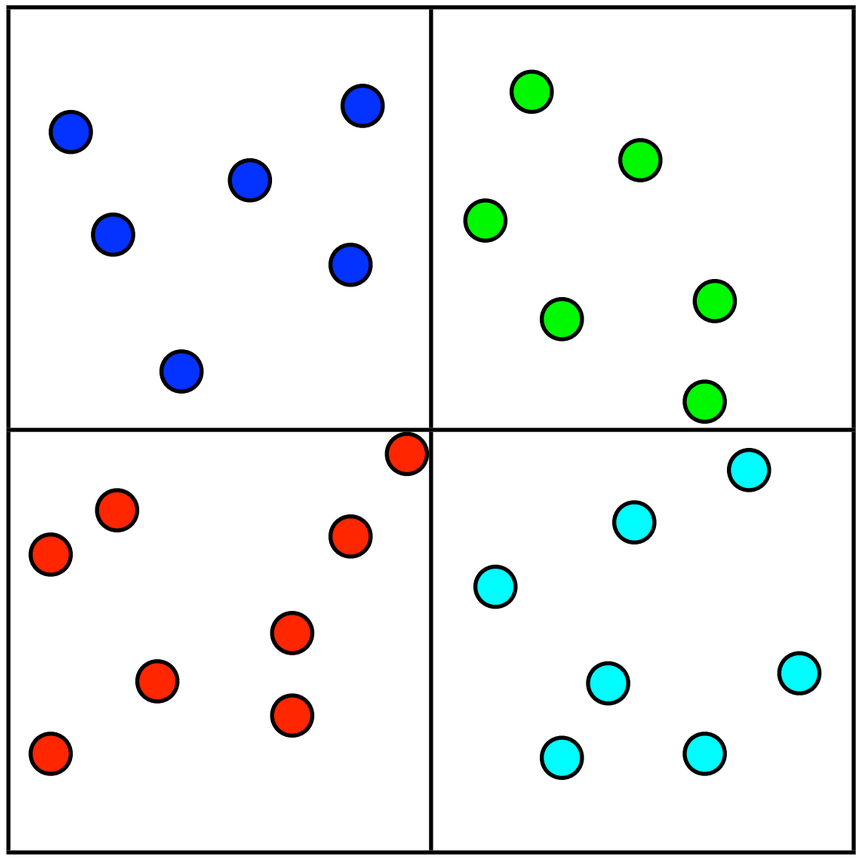
Implementation of the collision step

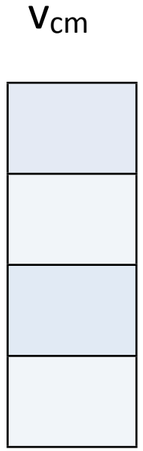
- The collision step is not as well suited for the GPU:
  - Calculating the center of mass velocity
    - is subject to read-modify-write problems
    - performs lots of random memory accesses
  - Rotating the particle velocities
    - needs random numbers for each collision cell
    - performs random reads (preferred) or writes

Calculating the center of mass velocity, the easy approach

- Using one thread per particle, atomic operations can be used to sum up velocities and masses for each cell
  - –Pros:
    - easy code
    - coalesced reads for particle data
  - –Cons:
    - very slow, due to 4 atomic operations per particle
    - no native atomic add for double precision
    - even slower when using atomicCAS-loop for double precision

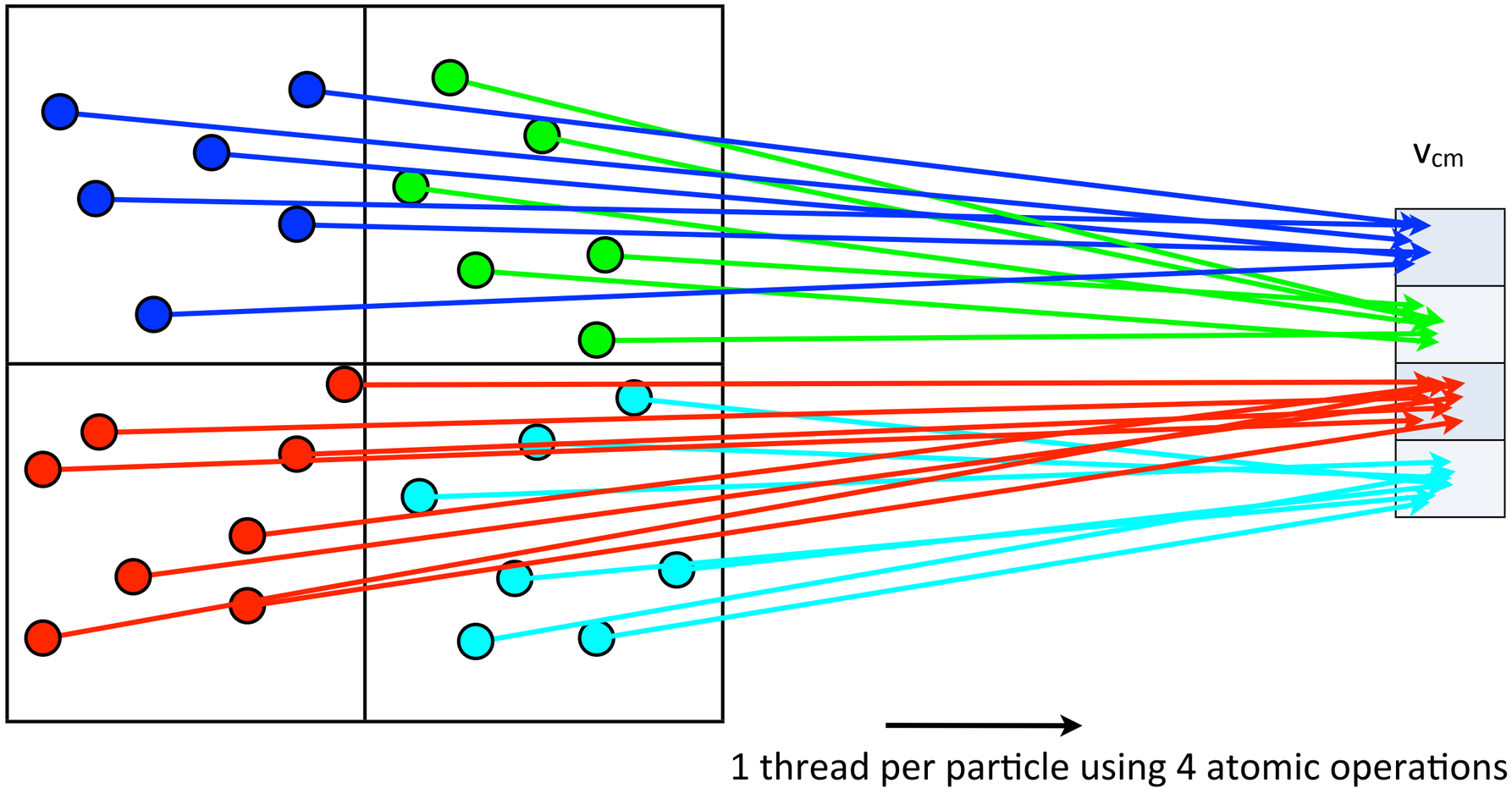
Calculating center of mass velocity, using atomic operations only





1 thread per particle using 4 atomic operations

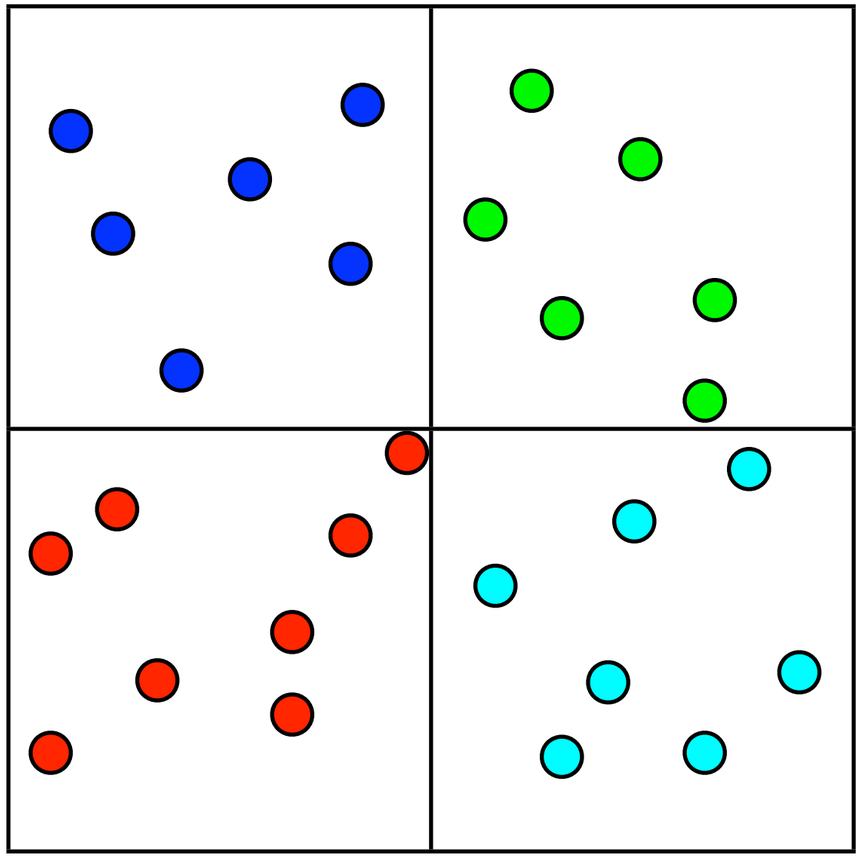
Calculating center of mass velocity, using atomic operations only



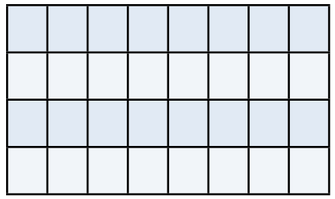
Calculating the center of mass velocity, the faster, but complicated way

- Build lists of particles for each cell
- Sum up velocities and masses according to cell lists
  - –Pros:
    - much faster, uses only one atomic operation per particle
  - –Cons:
    - complicated code
    - uses more memory for helpers
    - random reads for particle data
    - lists not reusable due to random shift of lattice

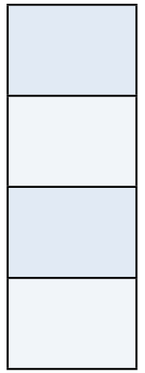
Calculating center of mass velocity using cell lists



Cell lists



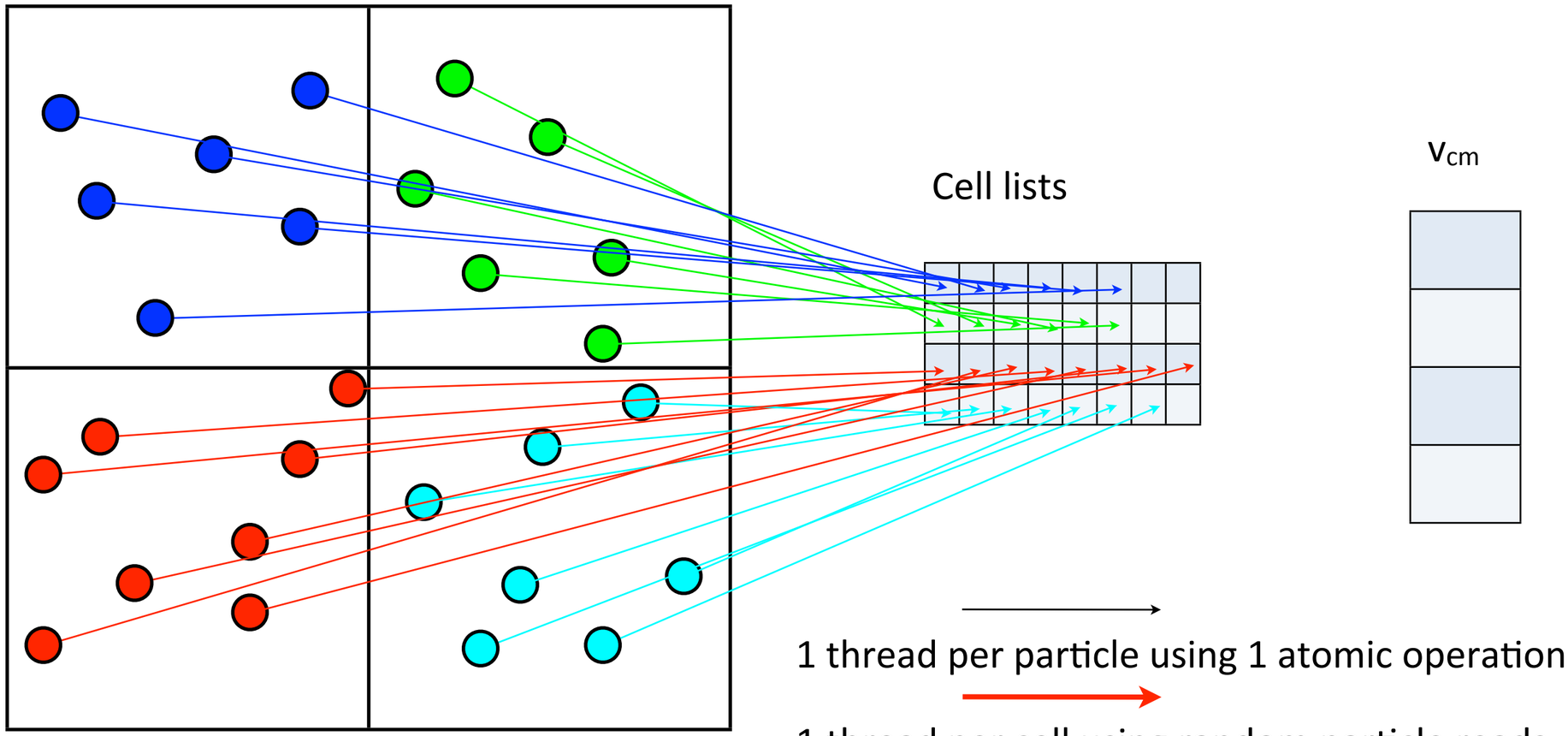
v_{cm}



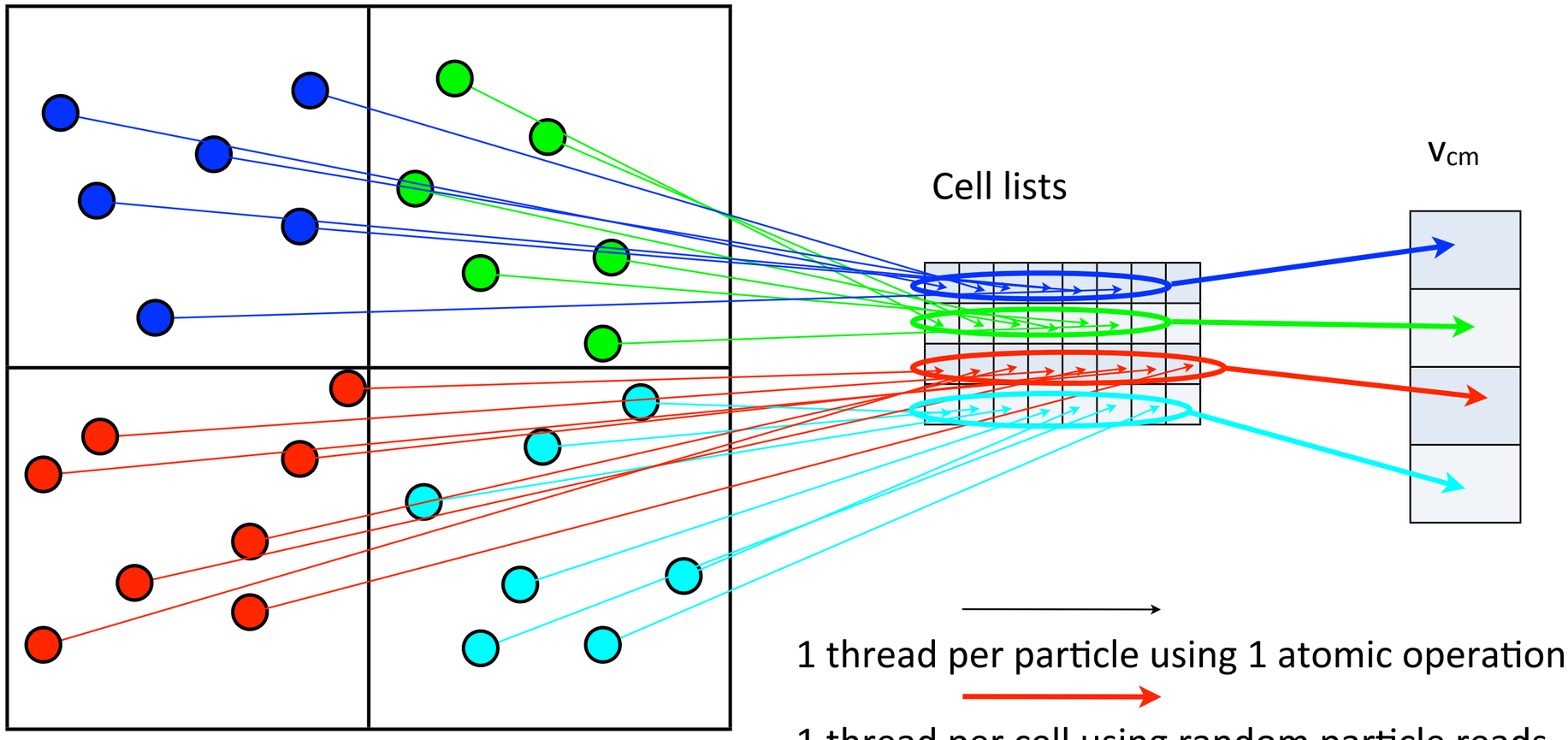
1 thread per particle using 1 atomic operation

1 thread per cell using random particle reads

Calculating center of mass velocity using cell lists



Calculating center of mass velocity using cell lists



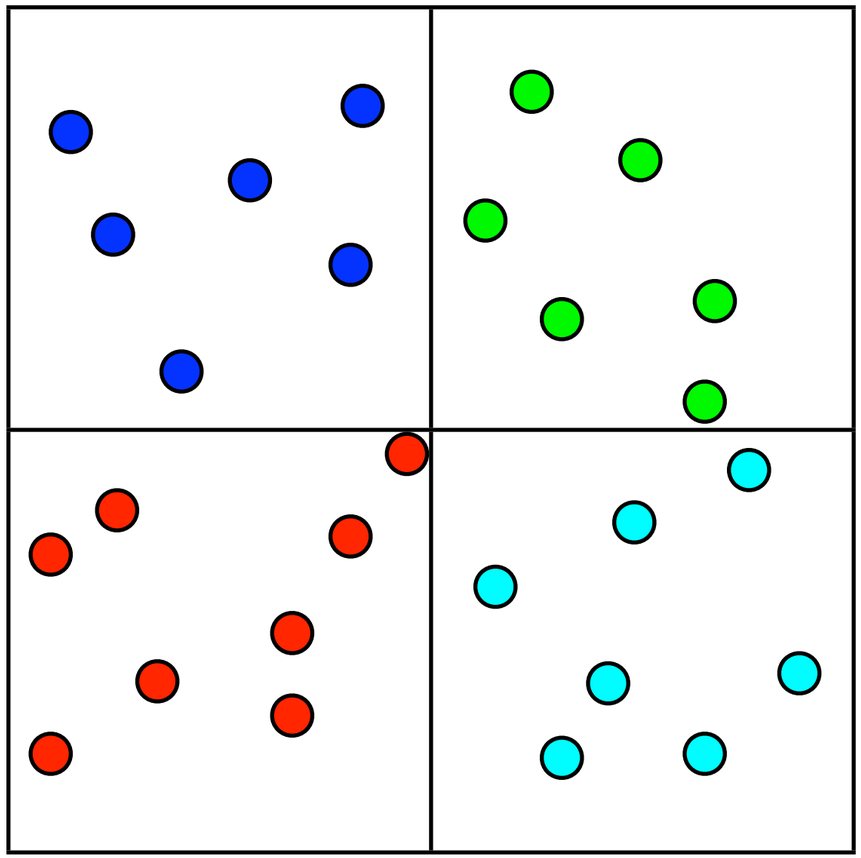
Calculating v_{cm}, building cell lists on GPU

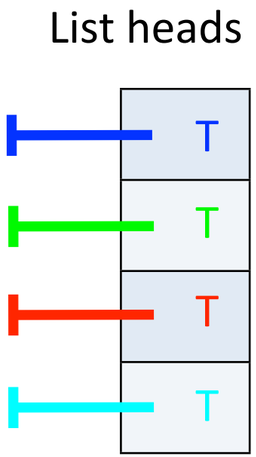
- Each particle has a reference to the next particle in its cell
- Each cell has a list head, initialized with a terminating value
- Thread configuration is one thread per particle

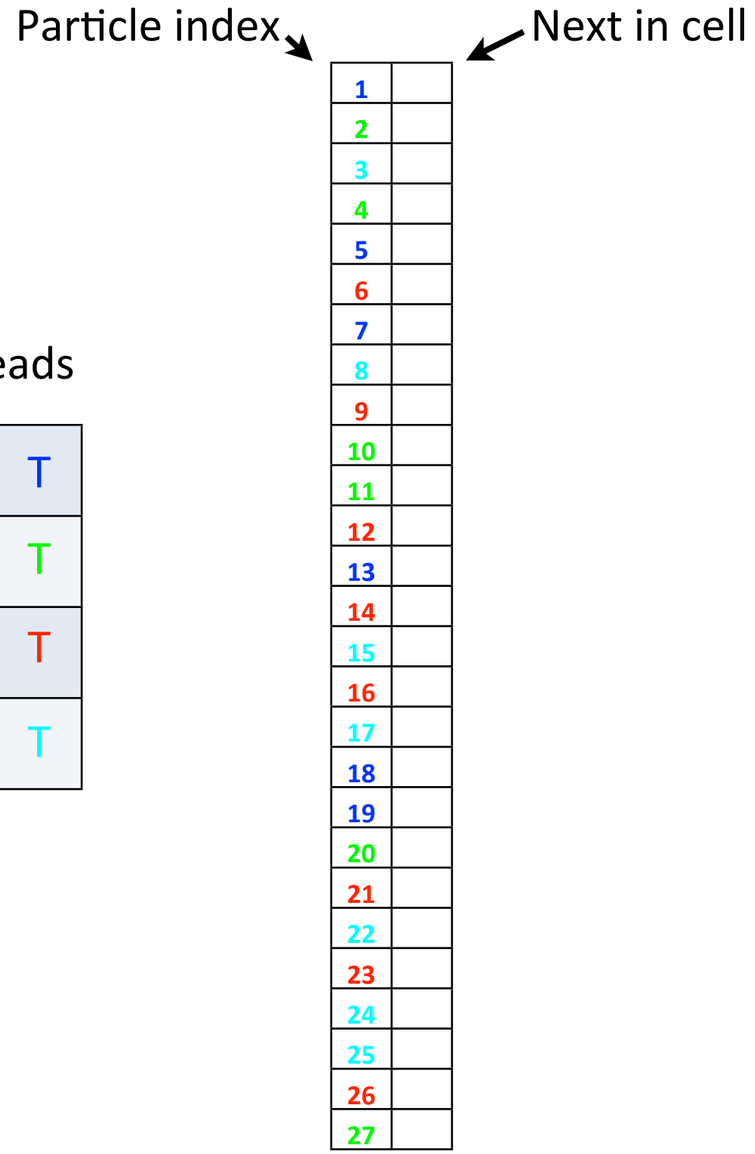
Kernel outline:

```
particle=GLOBAL_THREAD_INDEX;
cell=calculate_cell_index(particle);
next_in_cell[particle]=atomicExch(head[cell],particle);
/* Note: algorithm returns lists in descending order */
```

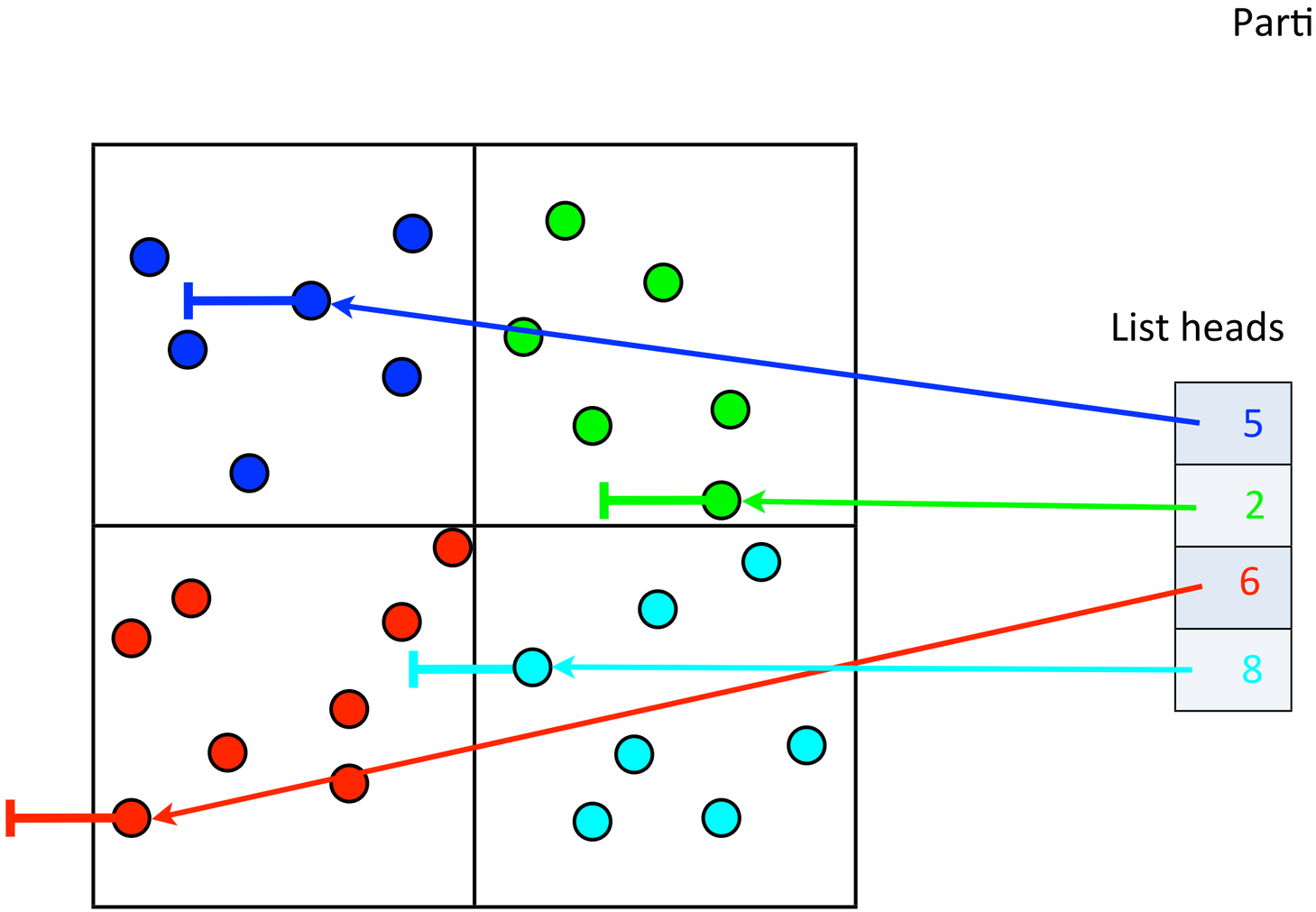
Calculating v_{cm}, building cell lists on GPU



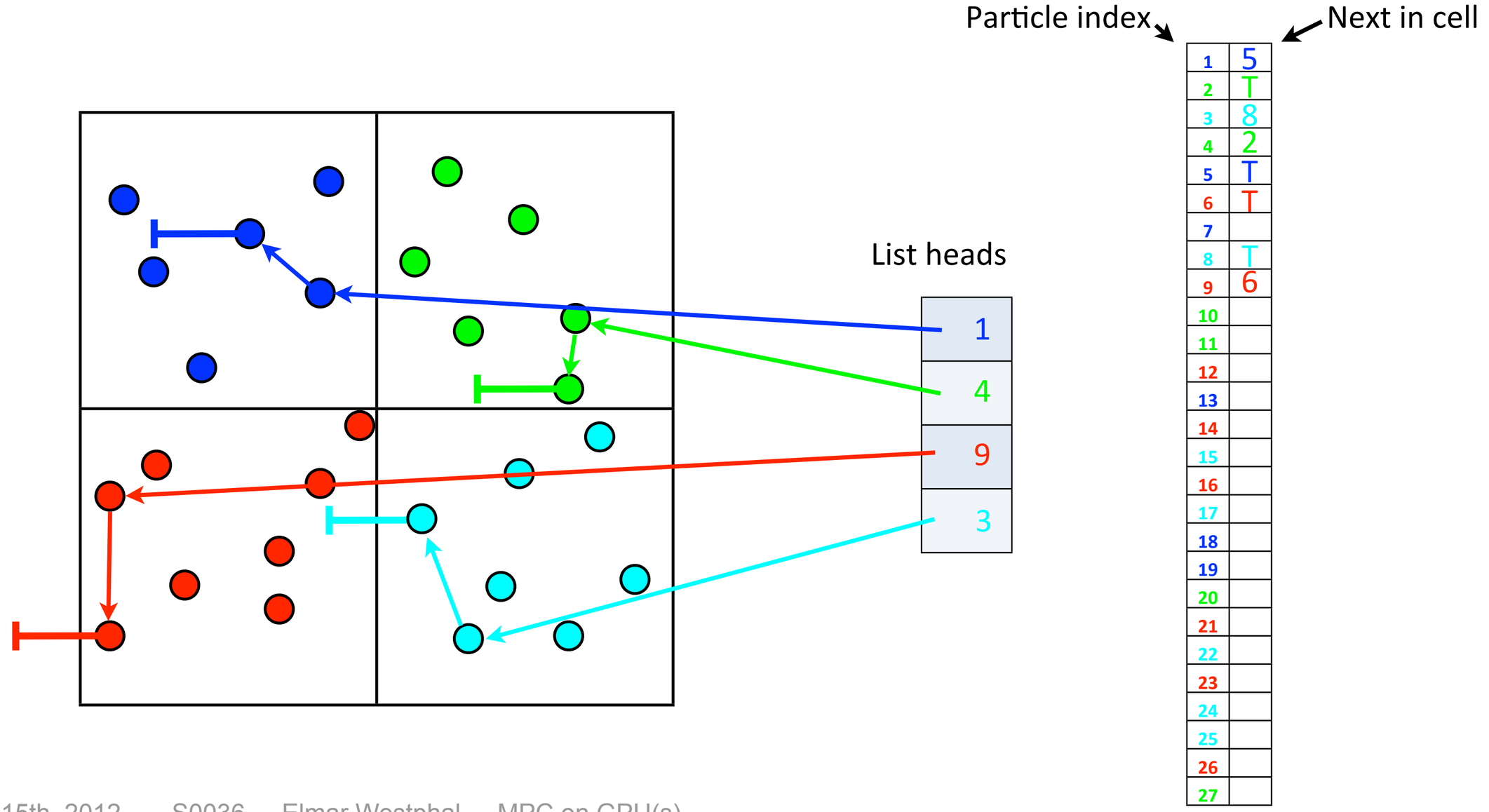




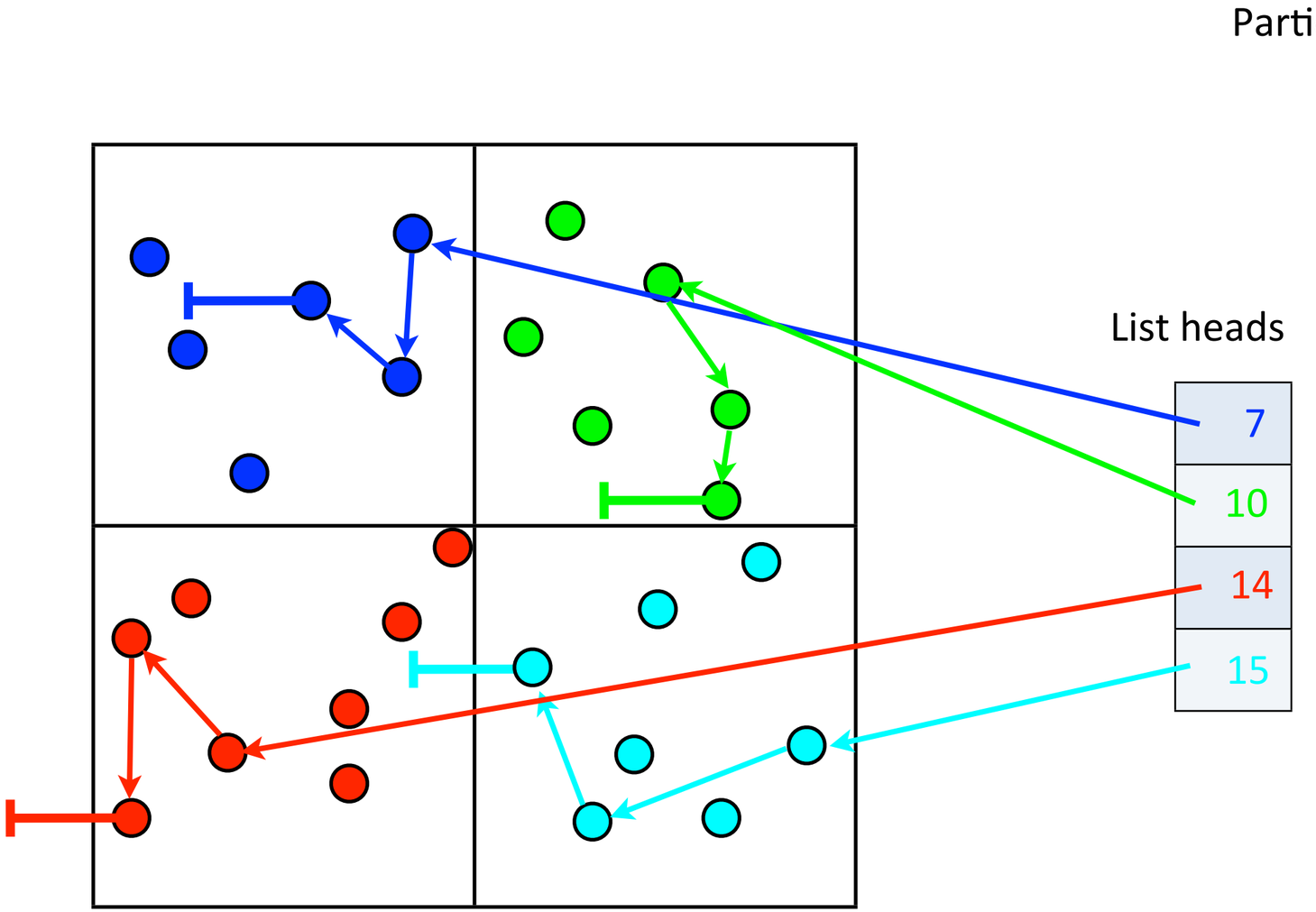
Calculating v_{cm}, building cell lists on GPU



Calculating v_{cm}, building cell lists on GPU



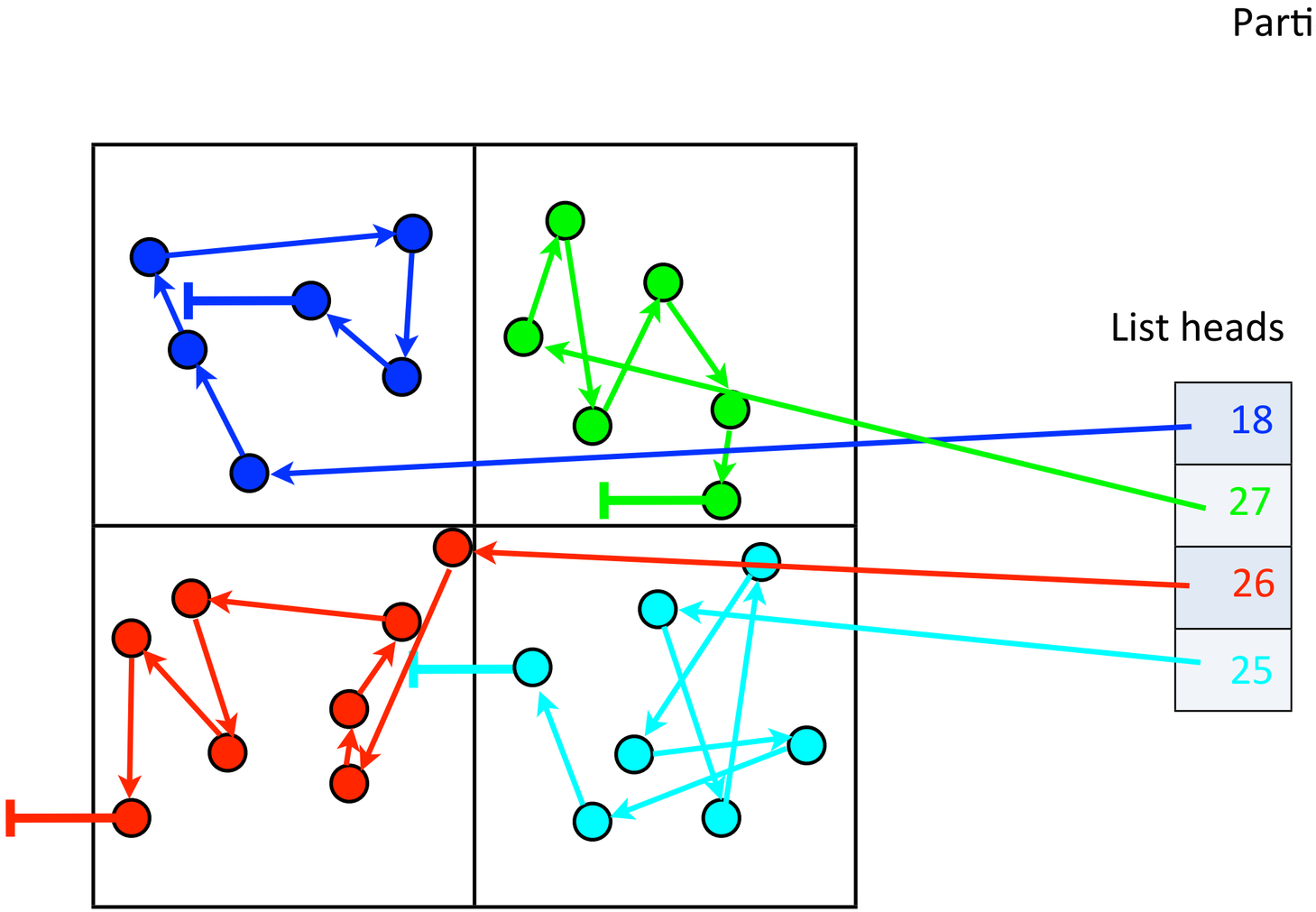
Calculating v_{cm}, building cell lists on GPU



Particle index → ← Next in cell

1	5
2	T
3	8
4	2
5	T
6	T
7	1
8	T
9	6
10	4
11	
12	
13	
14	9
15	3
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	

Calculating v_{cm}, building cell lists on GPU



Calculating v_{cm}, loop vs. tree-like reduction

Different techniques can be used to sum up the momenta:

- tree-like reduction schemes using one thread per particle
- summation loops using one thread per cell

Loops were chosen over trees because

- ρ_{fluid} is usually much smaller than the warp size
  - many threads are wasted
- ρ_{fluid} is mostly uniform over the simulation box
  - usually no big deviances in loop length

Rotation step

- 3 random number based key values for each cell's rotation matrix are pre-calculated
- It's cheaper to calculate the rest than to store/read it
- Data is processed particle-wise
  - cell-wise processing would cause random velocity writes

MPC on GPU(s): Optmizations

- Texture caches
- Streams for hiding data transfers
- Reordering fluid particles to improve data locality
- Building partial cell lists in shared memory
- Limiting the number of memory accesses

Texture cache

- Easy to implement
- Smaller than L1-cache, but can be employed as needed
- Texture fetch has 128 bits payload, regardless of type:
  - cell properties are stored in pairs of doubles (2x64 bits, implemented using int4 textures) to reduce texture fetches
- Larger cache lines of L1 cache may even slow down performance

Streams for hiding data transfers

- Solute particle data are transferred in every MPC step
- CUDA streams can do this parallel to calculations:
  - import of velocities and positions happens during fluid integration
  - export of velocities happens during fluid particle rotation
- At ρ_{fluid}=10, the transfer of ~0.7 solute particles/cell can be completely hidden, enough for our typical use cases

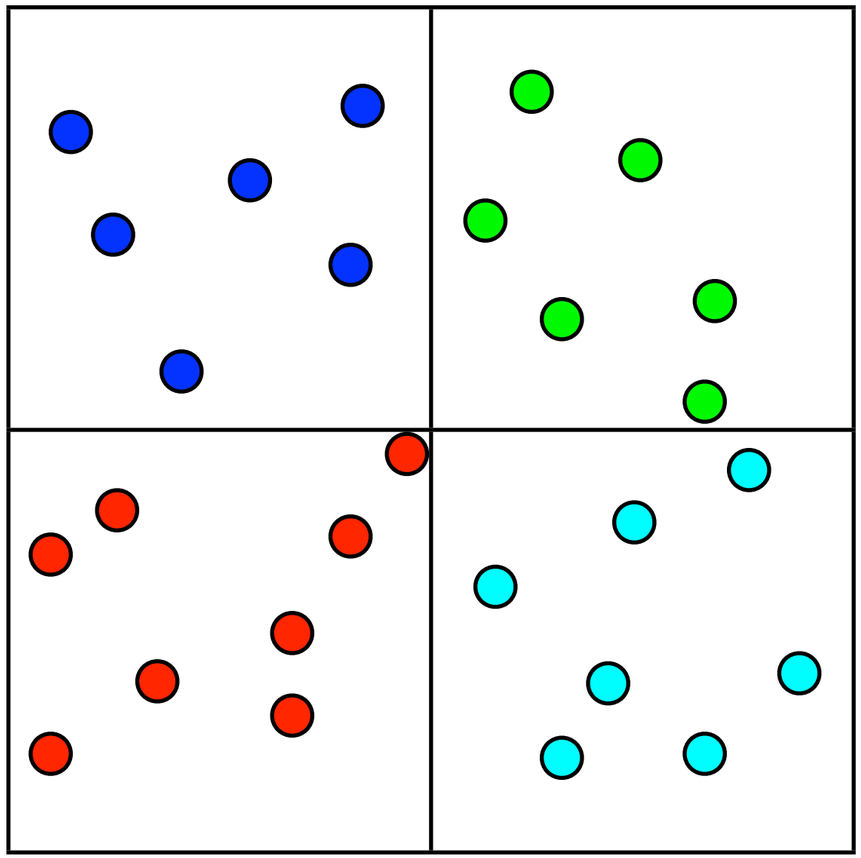
Reordering of fluid particles to improve data locality

- (Texture) caches help accelerating random particle and cell property reads, but
  - caches are small
  - the amount of data is huge
  - caches help best if there is data locality
- Our typical access pattern for seemingly random access is “cell-wise”, so reordering should be done this way

Reordering schemes

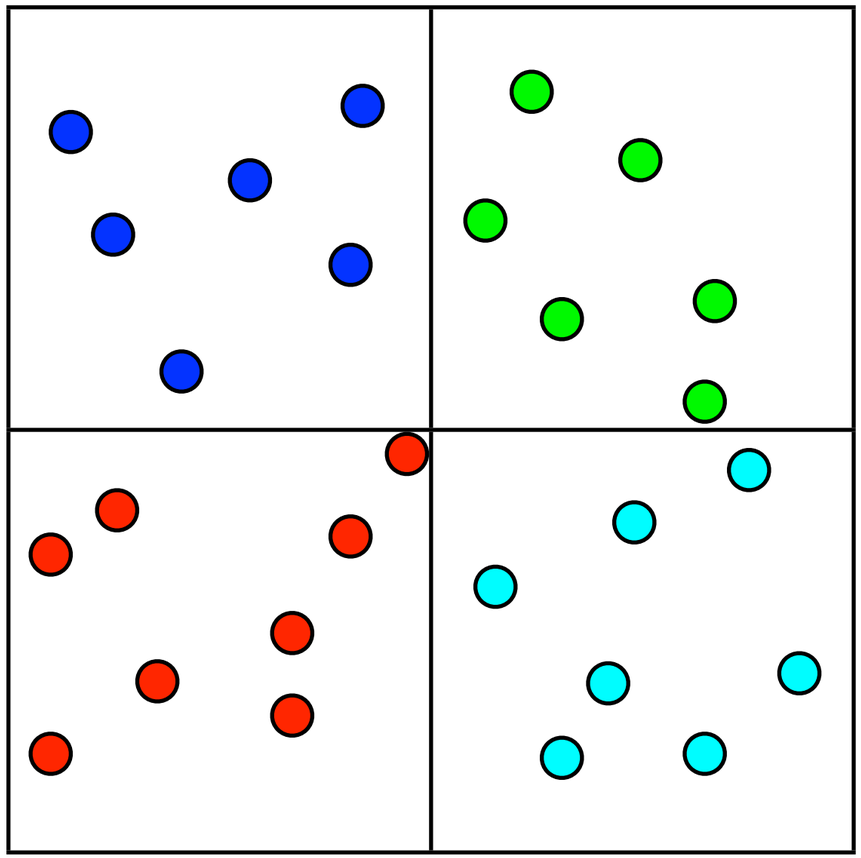
- Cell-wise reordering may be done in different orders:
  - according to the polynomial (xyz-coordinates) cell index
  - along room filling curves:
    - Morton curves (also known as Z-curves)
    - Hilbert curves etc...
- In this code, there was no significant performance difference between polynomial indices and Z-curves
  - polynomial indices are chosen for easier implementation
  - other optimizations are easier to apply on polynomial indices

Calculation of v_{cm} on unordered particles

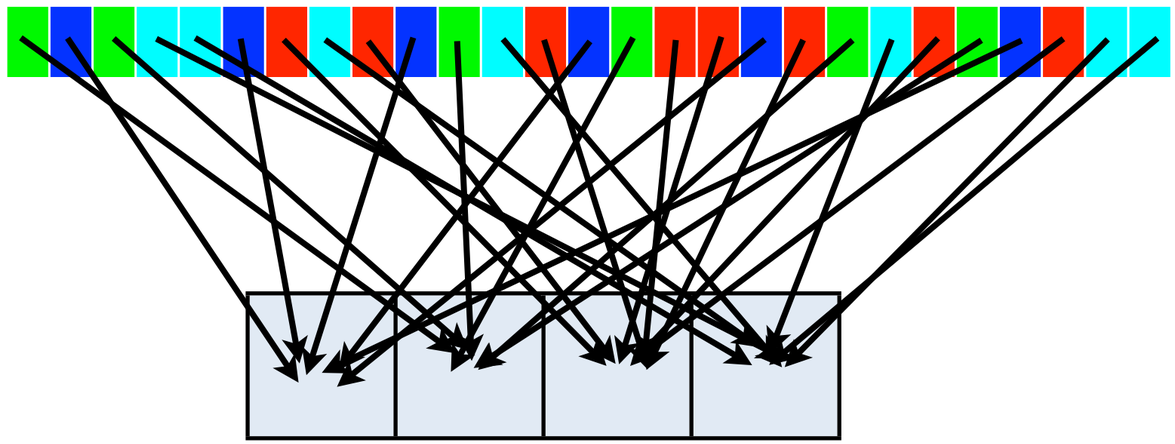


Placement in memory

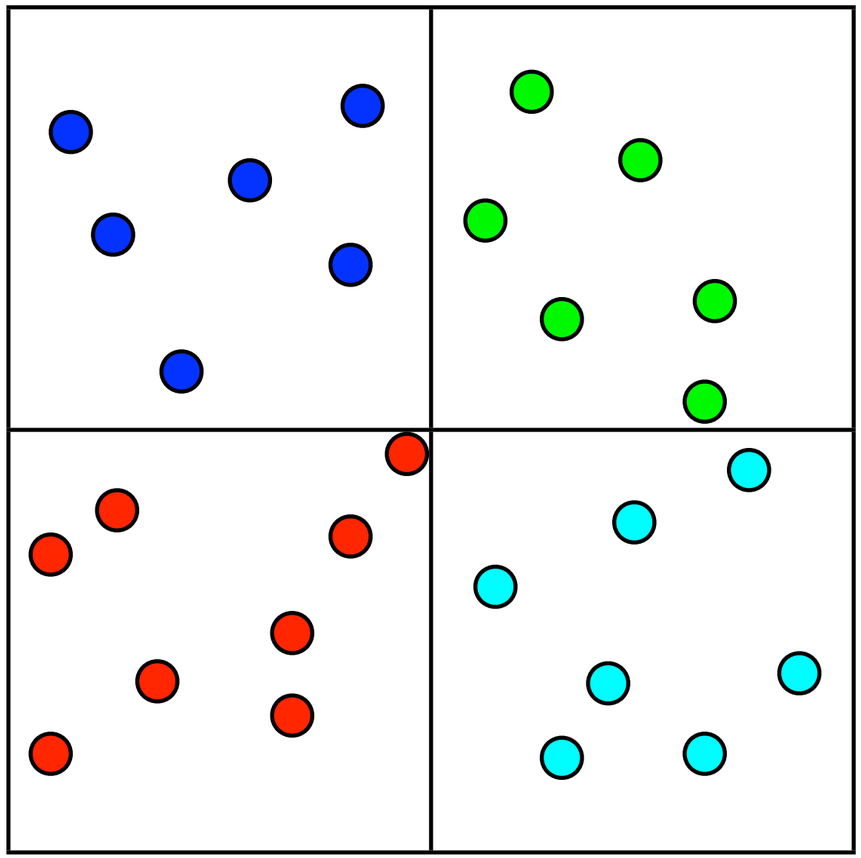
Calculation of v_{cm} on unordered particles



Placement in memory

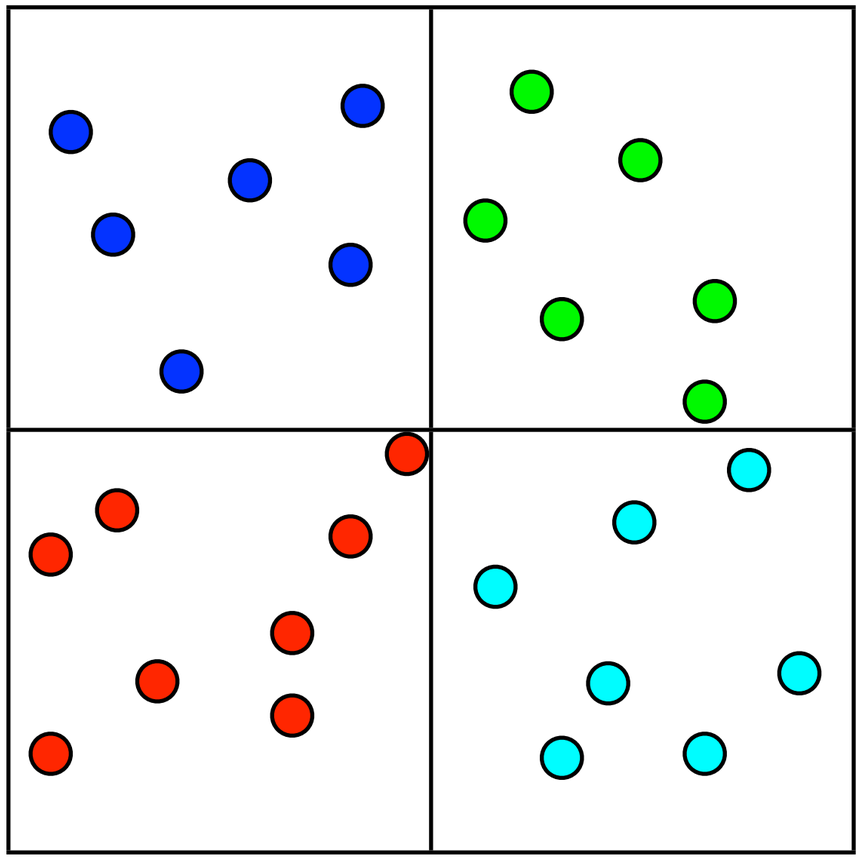


Calculation of v_{cm} on unordered particles



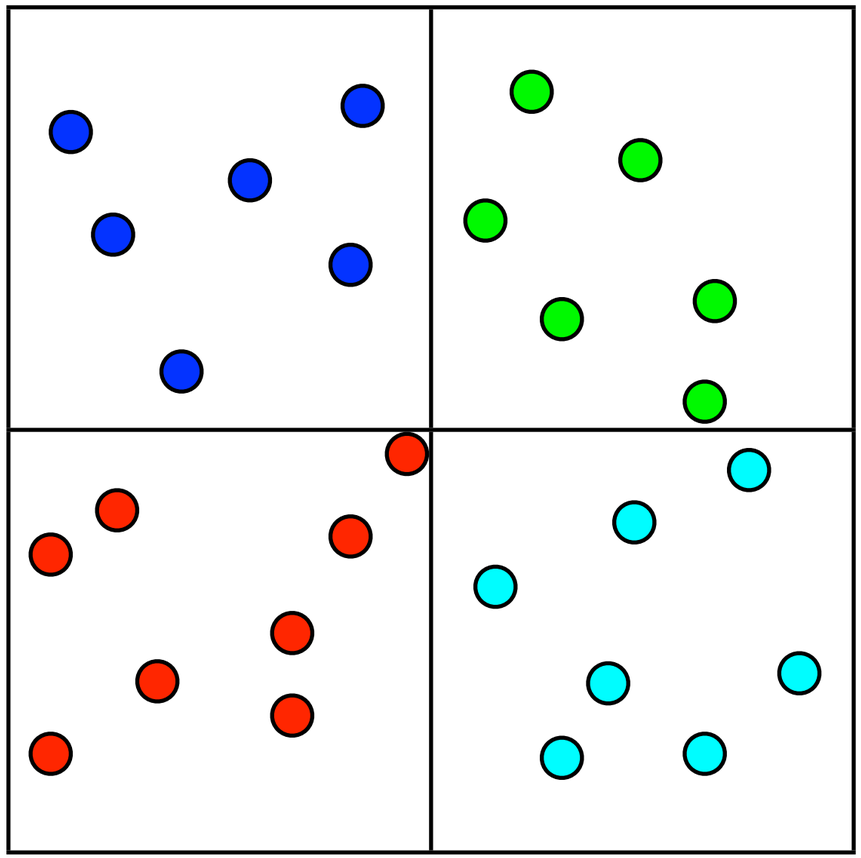
Placement in memory

Reordering and calculation of V_{cm}



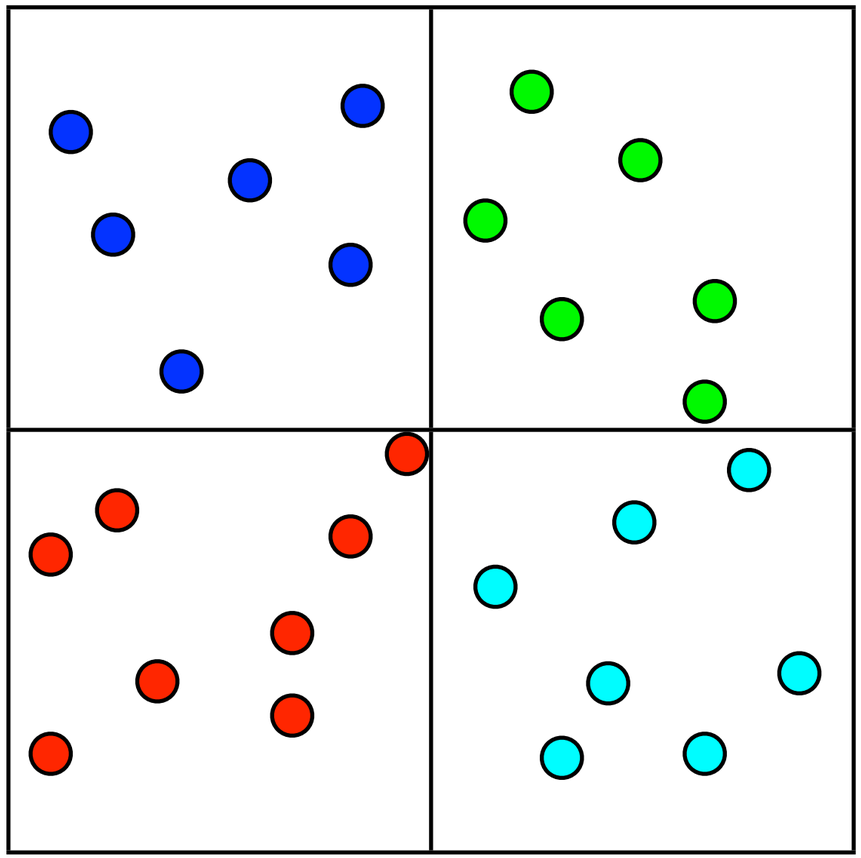
Placement in memory

Reordering and calculation of V_{cm}

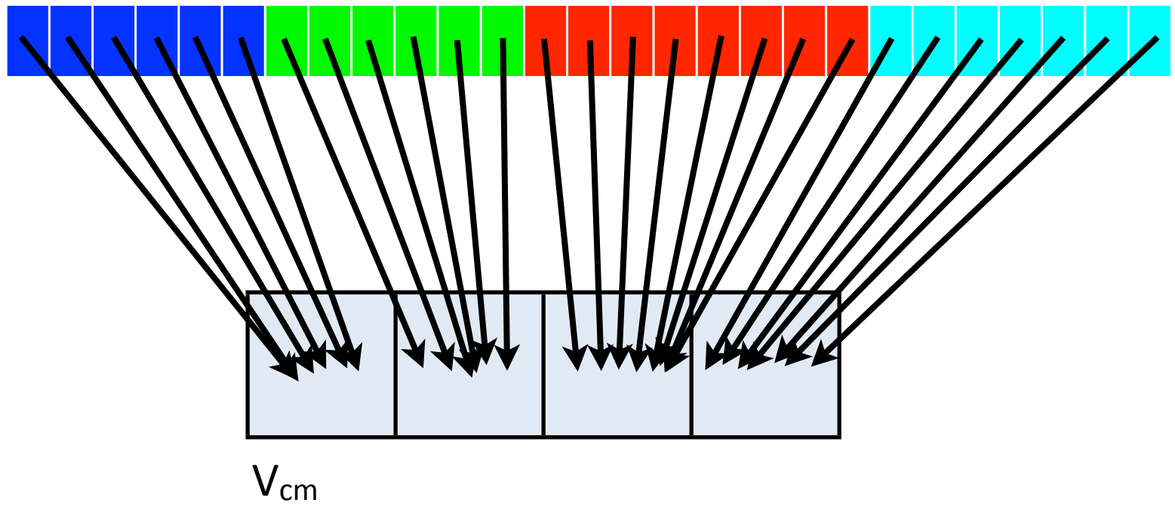


Placement in memory after reordering

Reordering and calculation of V_{cm}



Placement in memory after reordering



Building partial cell lists in shared memory

- In a system with N_C collision cells, shared memory is usually not big enough to store N_C lists
- In a threadblock with N_B thread using one particle per thread, we at most access N_{block} <u>different</u> lists
- N_C partial lists can be (hash-)mapped to N_B slots
- In parallel, computing the hash index is subject to read-modify-write problems

Building partial cell lists in shared memory, cont.

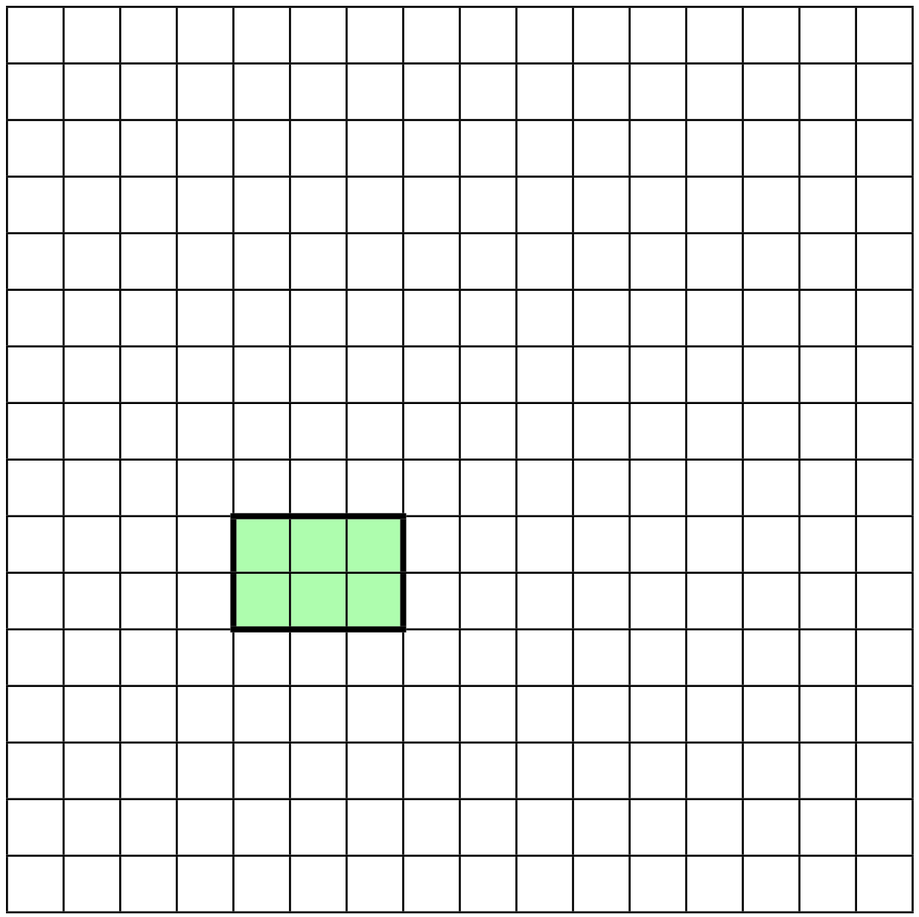
- (at least) 2 atomic operations in shared memory per particle:
  - atomicCAS to determine the hash index, may be repeated
  - atomicExch for inserting the particle into the partial list
- one atomicExch on device memory per partial lists for concatenation to final result
- Gain only if partial list length is greater than 1:
  - usually (only) works in somewhat recently ordered systems

Building partial cell lists in shared memory, cont.

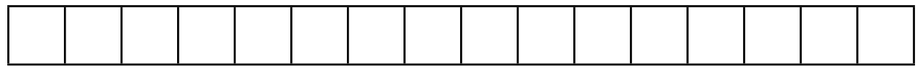
```
do {
 ++count;
 hi = potential_hash_index(my_cell, count) % BLOCKSIZE;
 res = atomicCAS(shared_hashes + hi, -1, my_cell);
} while (res != -1 && res != cell);
next_in_list = atomicExch(shared_heads + hi, myThread);
```

- `res` equals:
  - -1 if the hash value was not yet taken (done)
  - my_cell if it was taken by the same cell index (done)
  - something else in case of a hash collision (repeat)

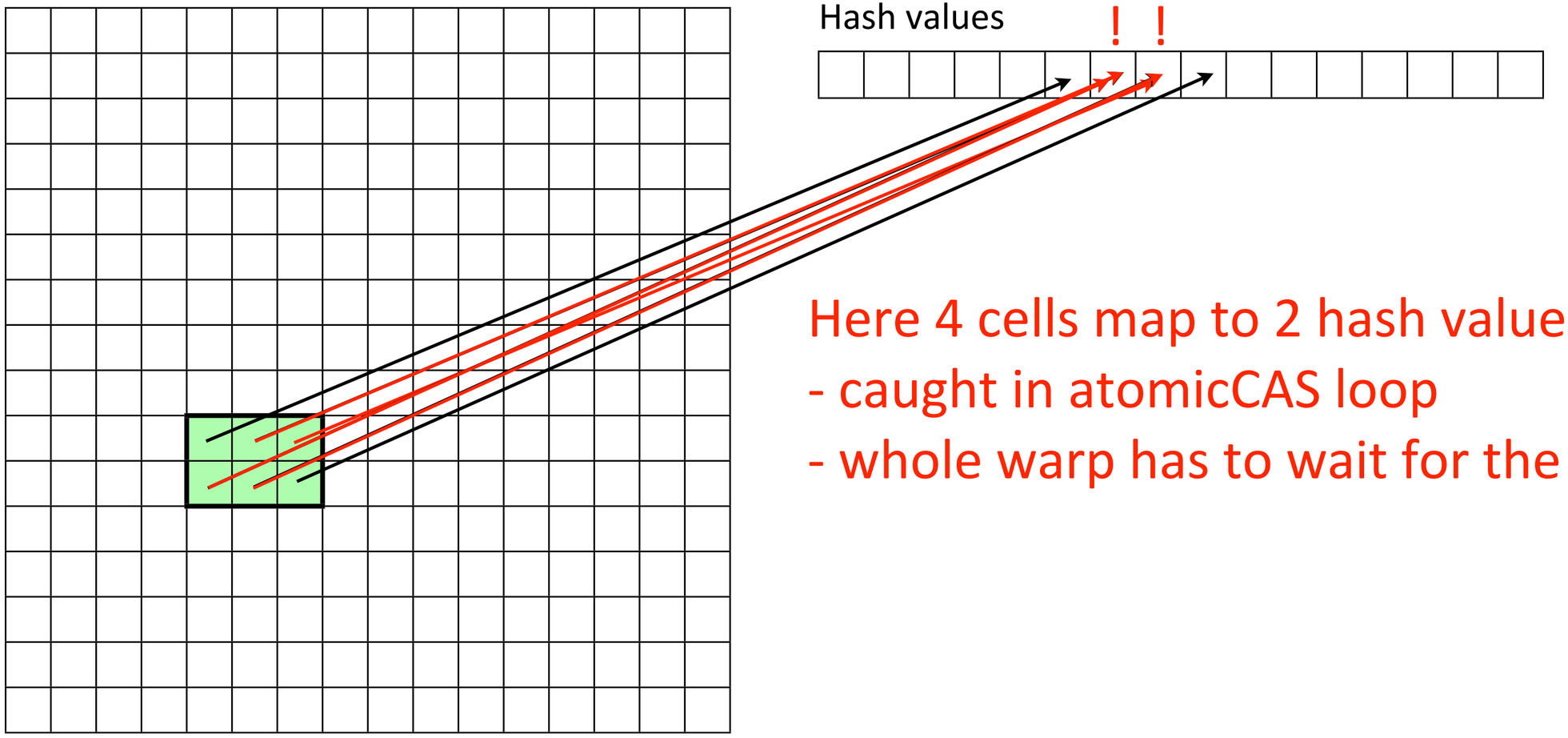
Problem: hash collisions stall the whole warp



Hash values

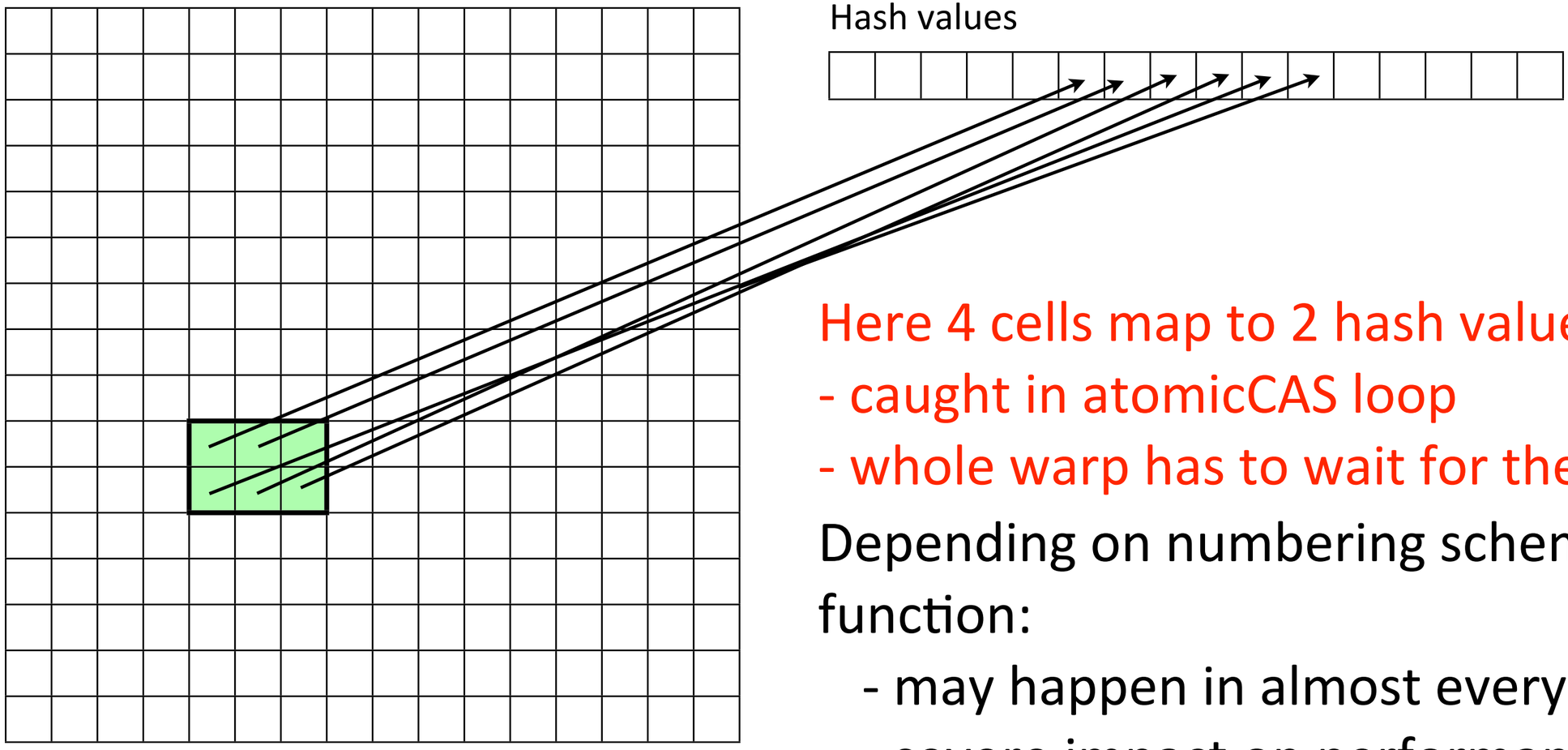


Problem: hash collisions stall the whole warp



Here 4 cells map to 2 hash values:
- caught in atomicCAS loop
- whole warp has to wait for the loop

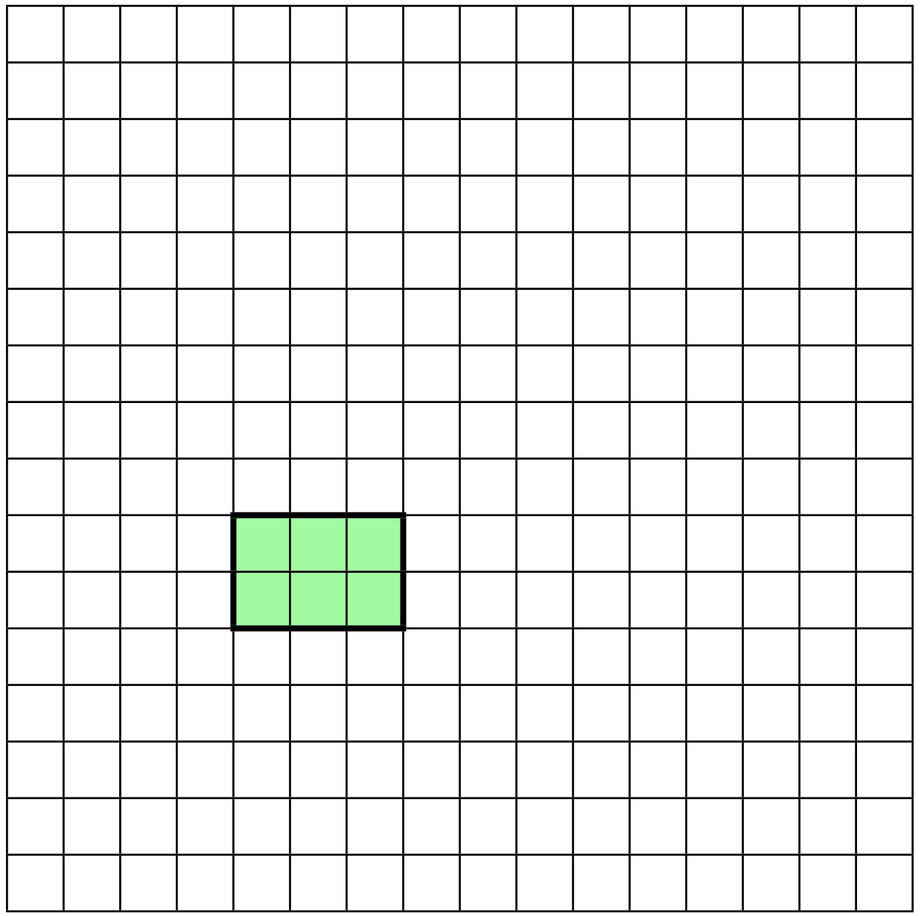
Problem: hash collisions stall the whole warp



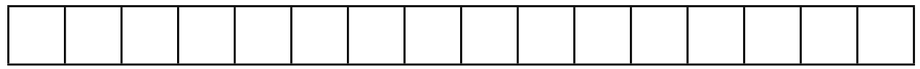
Here 4 cells map to 2 hash values:
- caught in atomicCAS loop
- whole warp has to wait for the loop

Depending on numbering scheme and hash function:
- may happen in almost every warp
- severe impact on performance

Solution: geometry aware hash functions



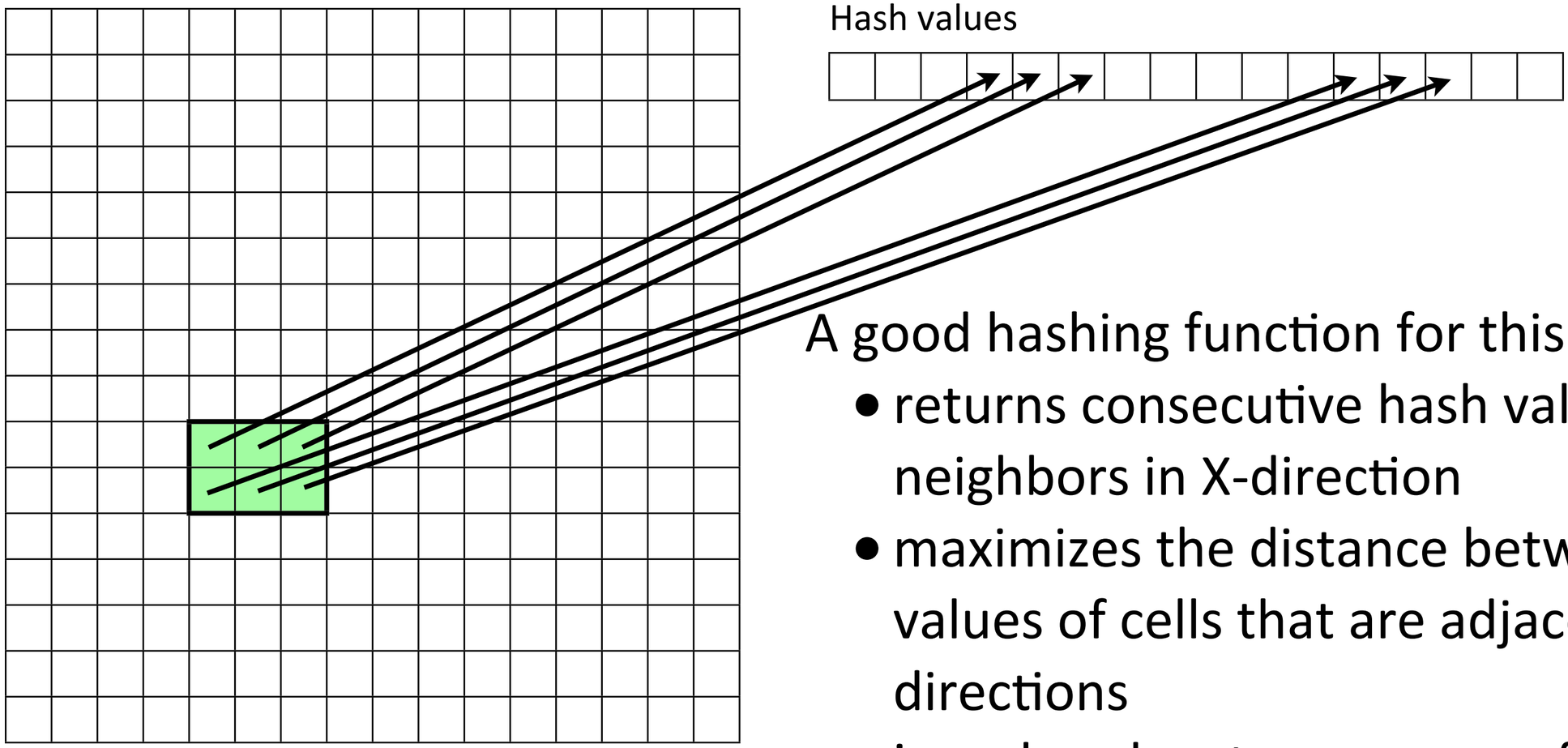
Hash values



A good hashing function for this problem (e.g.)

- returns consecutive hash values for neighbors in X-direction
- maximizes the distance between the hash values of cells that are adjacent in other directions
- in ordered systems, range of cell is (somewhat) predictable

Solution: geometry aware hash functions



A good hashing function for this problem (e.g.)

- returns consecutive hash values for neighbors in X-direction
- maximizes the distance between the hash values of cells that are adjacent in other directions
- in ordered systems, range of cell is (somewhat) predictable

Limiting the number of memory accesses and including additional functionality

- Memory access is much more expensive than calculation
- Parts of the algorithm are combined to reduce the number of reads (e.g. building cell list during integration)
- A usually expensive cell-wise thermostat shares data and instructions with other functions and can be “inlined”
- Also, shear flow is added with very little overhead

Additional functionality: thermostat

- Implemented as a per-cell thermostat
- Needs kinetic energy per cell, expensive to calculate, but
- Kinetic energy is calculated from the same data as v_{cm}
- Velocities are scaled by energy-dependent random factor
- This can be applied during rotation step
- Noticeable overhead: about 10% for
  - calculation of one random number per cell
  - writing/reading of correction factor

Additional functionality: shear flow

- Adjustment are applied during integration step
- Particles crossing upper/lower y-boundary are subject to
  - x-position shift depending on simulation time and shear rate
  - x-velocity adjustment depending on box size and shear rate
- Crossing can be caused by
  - Particle movement (adjustment are permanent)
  - Shift of lattice (undone in extra kernel after MPC step)
- Results in a sheared velocity profile (after some time)
- Total overhead: ~5%

Floating point precision

- (almost) all calculations are performed in double precision
- All velocities and intermediary results are stored in double precision to improve energy and momentum conservation
- Fluid particle positions are subject to random lattice shift, so they can be stored in single precision
- Solute particle data usually come from external code and are expected in double precision
- Code can be compiled single precision only for older devices

Memory usage

per Particle		per Cell	
Particle position	3*4 bytes (3*8 for solute)	Center of mass velocity	3*8 bytes
Particle velocity	3*8 bytes	Rotation matrix	3*8 bytes
Cell & cell list	2*4 bytes	Random seed	8 bytes
Helpers	8 bytes	Cell list head	4 bytes
Particle total	52 (64) bytes	Cell total	60 bytes

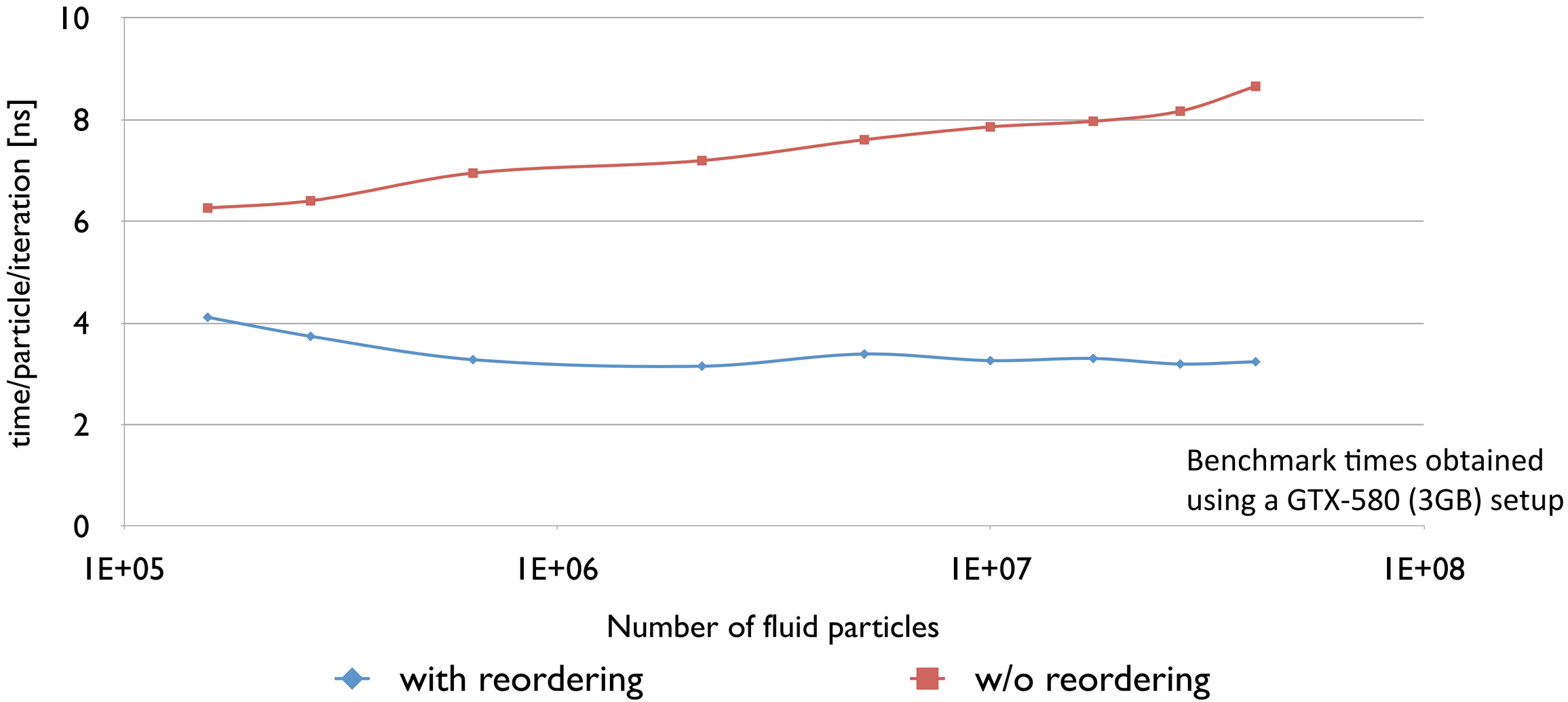
For 1 cell with ρ_{fluid}=10 and ρ_{solute}=1 : 60 + 10*52 + 64 = 644 bytes

Achievable system sizes

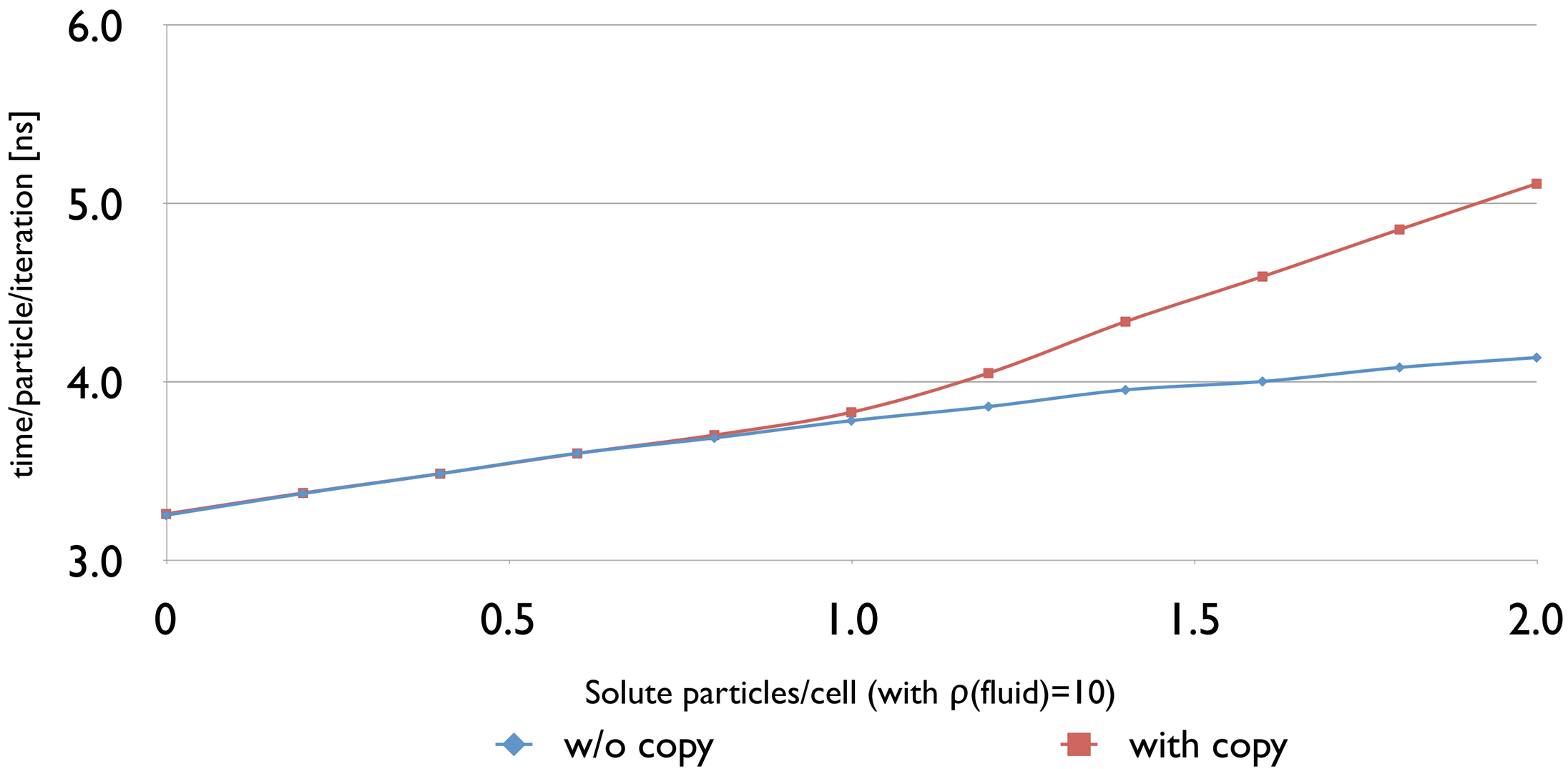
This yields the following maximum sizes for our simulation boxes:

GPU Memory	Max. size for cubic system
1.5 GB	~128^3
3.0 GB	~163^3
6.0 GB	~208^3

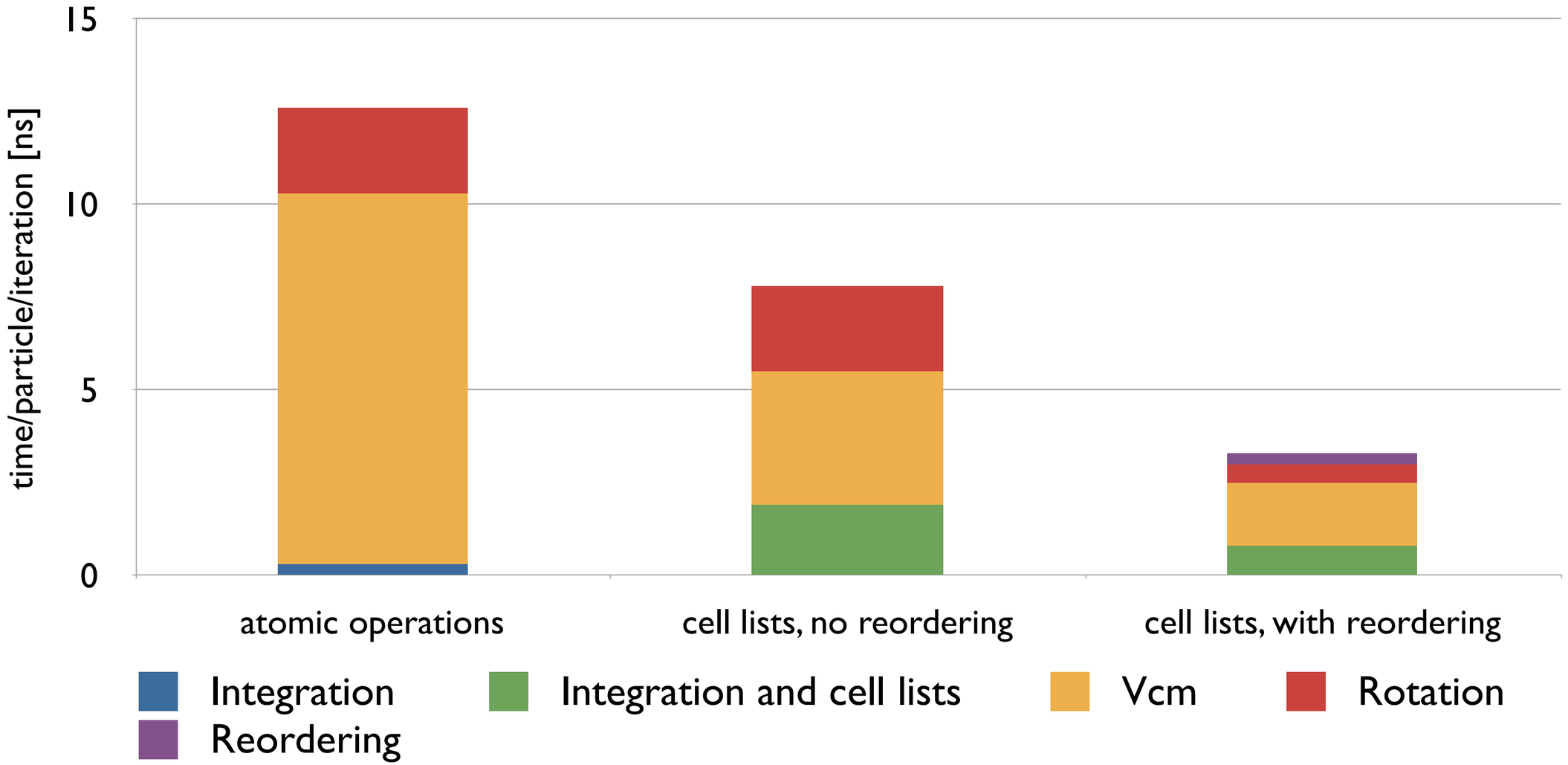
Normalized runtimes for different system sizes



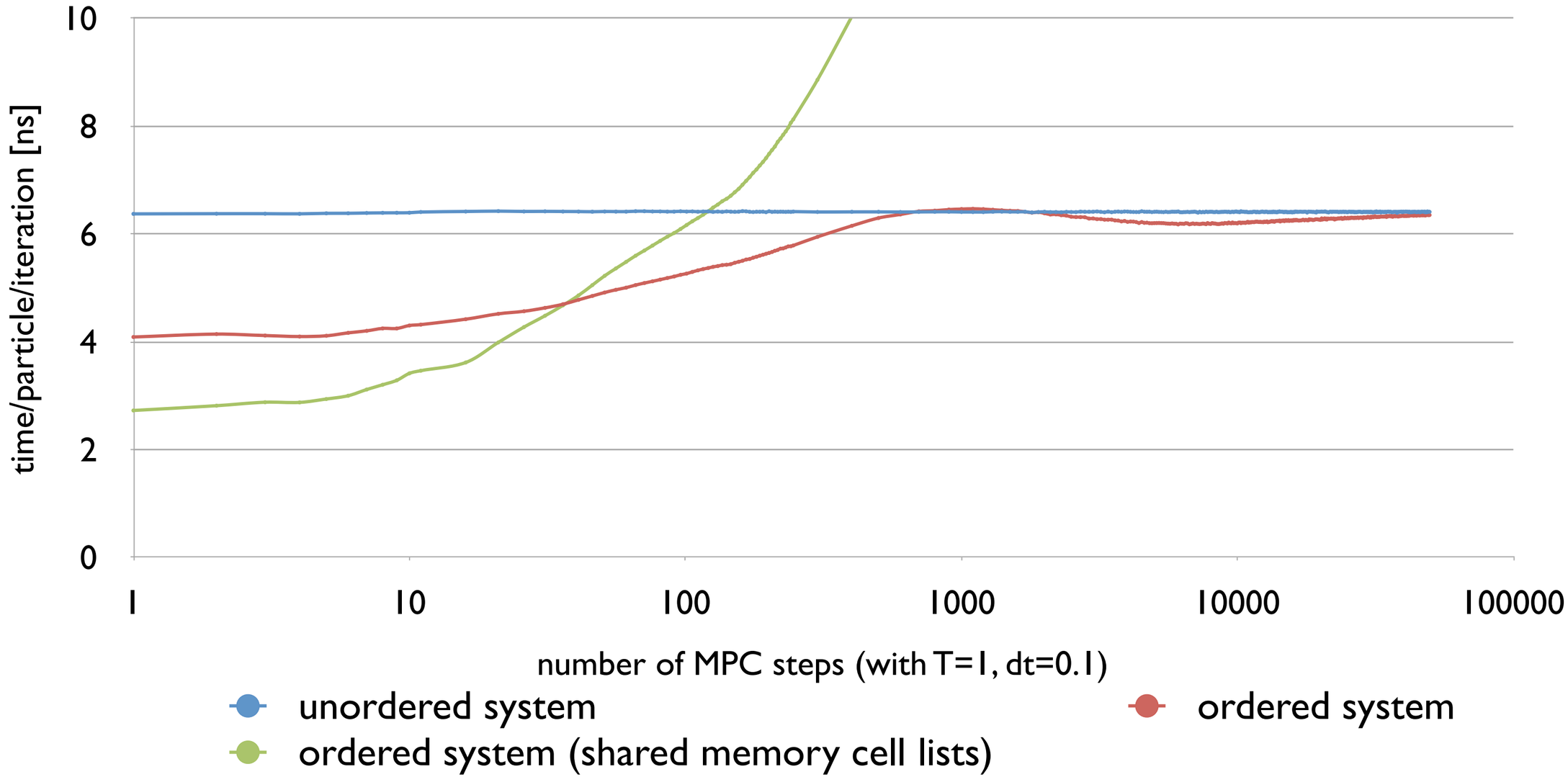
Normalized runtimes for different solute densities



Partial times for different implementations



Runtime behavior after reordering



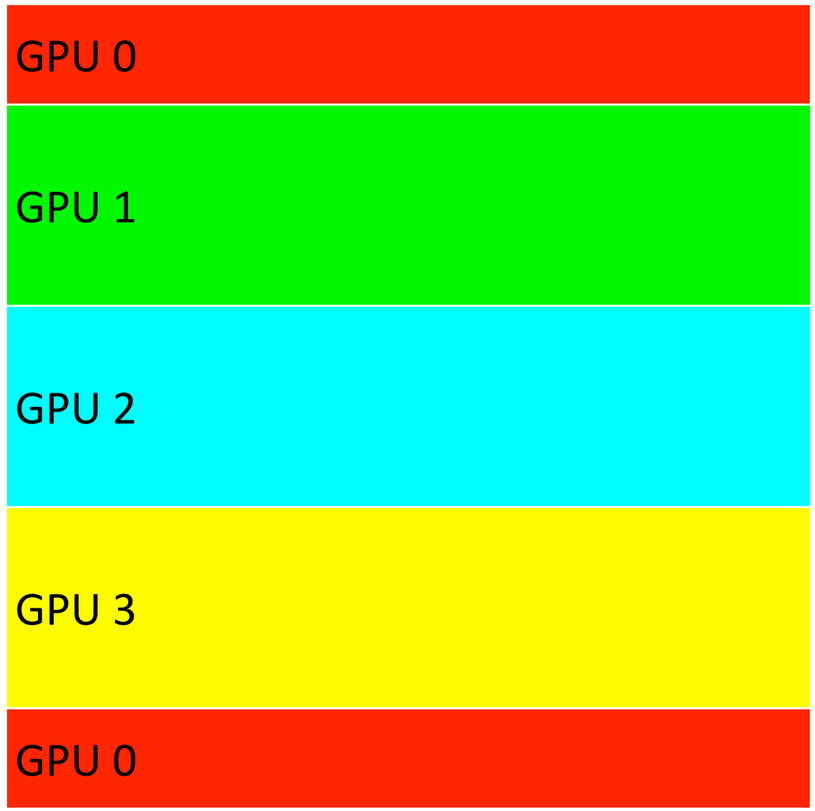
Multi-GPU implementation

- Going multi-GPU
- Domain decomposition
- Halo areas
- Solute particle handling
- Optimizations
- Benchmark results

Going multi-GPU

- Code supporting multiple GPUs enables us to increase system sizes and/or simulation speed
- CUDA 4 made multi-GPU projects significantly easier
- The code developed is a single-host multi-GPU implementation combining CUDA 4 and OpenMP
- A proof of concept application scales well on machines with 4 GPUs (tested on dual GTX-590 and quad GTX-580)

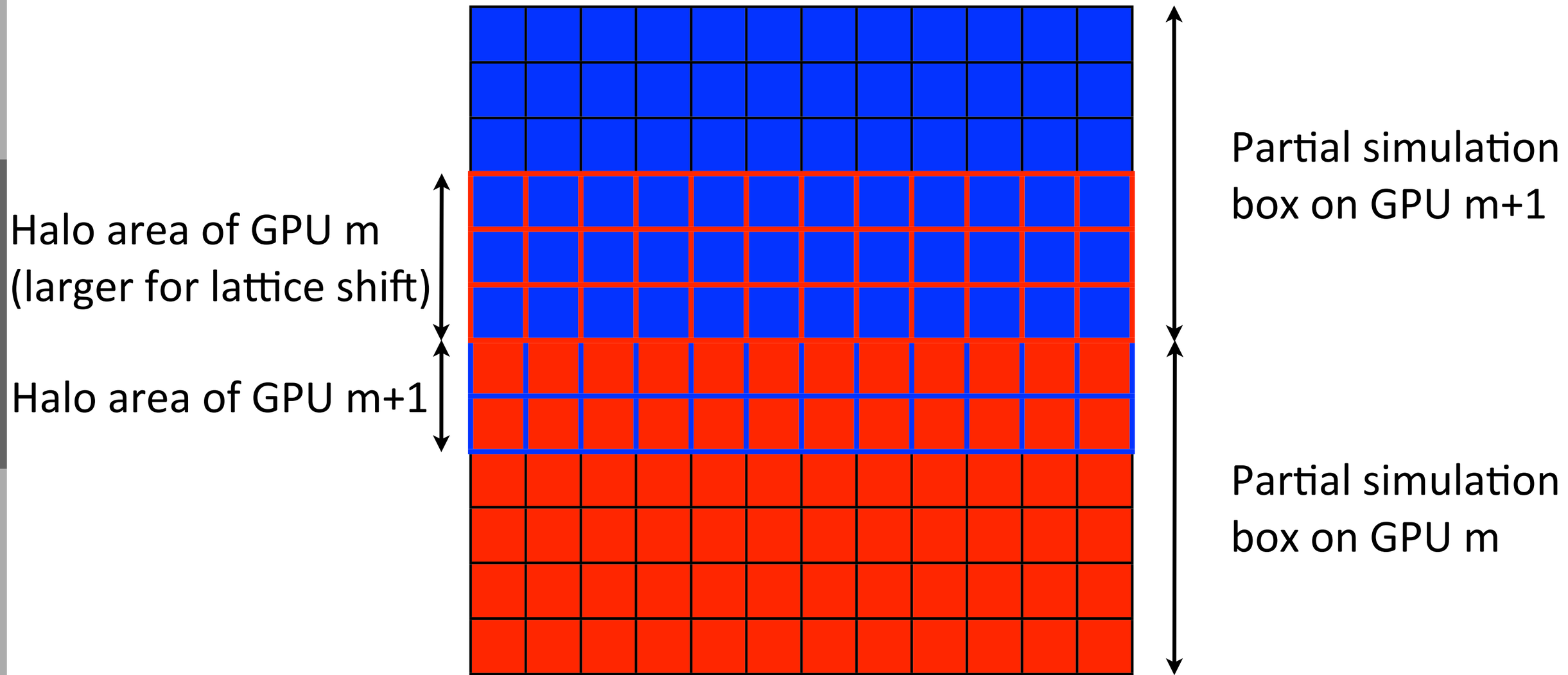
Domain decomposition and workload sharing



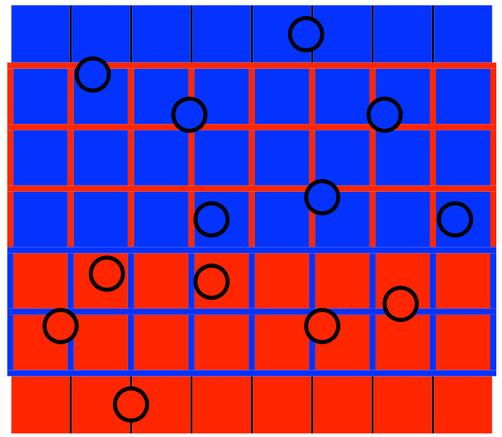
Domain decomposition for 4 GPUs

- For n GPUs, the simulation box is divided into n slices of similar size:
  - the slices are cut parallel to the x-z-plane
  - the slice of GPU 0 wraps around the upper/ lower y-boundary for easier implementation of periodic boundary conditions and shear flow
  - solute particles are primarily handled by GPU 0

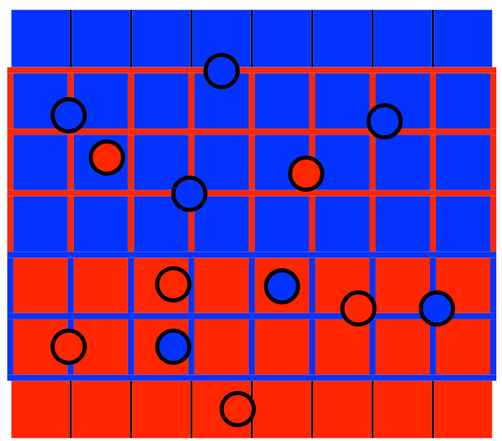
Halo areas between sub-boxes on different GPUs



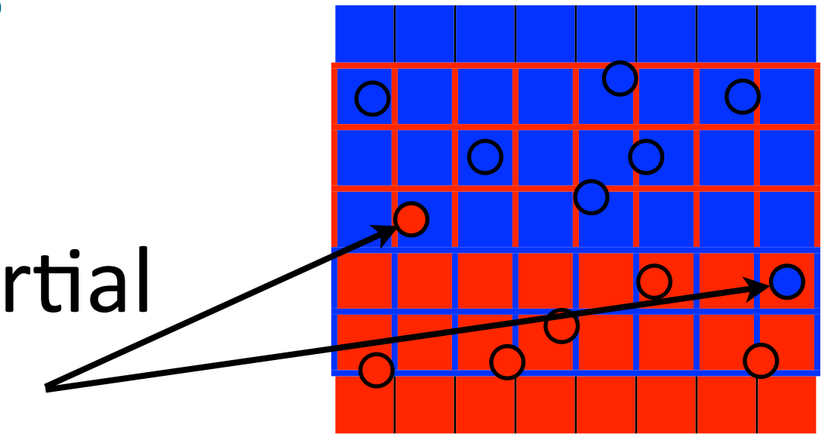
Fluid particle handling in halo areas



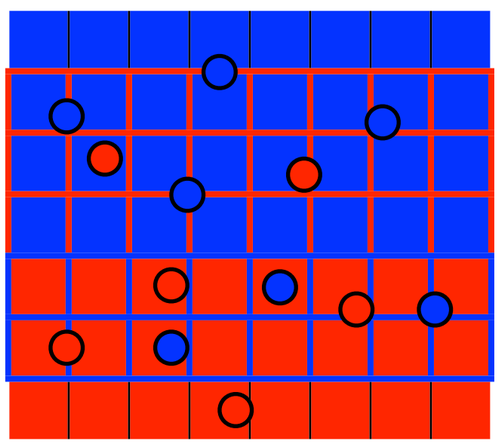
- Particles moving out of their GPUs partial box enter a 2-3 cell wide halo area
- The upper halo area is larger to accommodate lattice shift
- In border areas, intermediary results are exchanged between GPUs to calculate v_{cm}
- When a particle exits the halo area, particles are redistributed over the GPUs



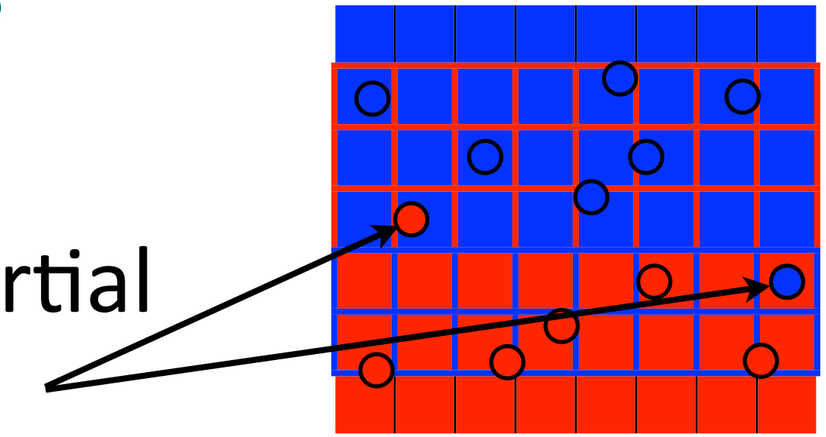
Fluid particle handling in halo areas



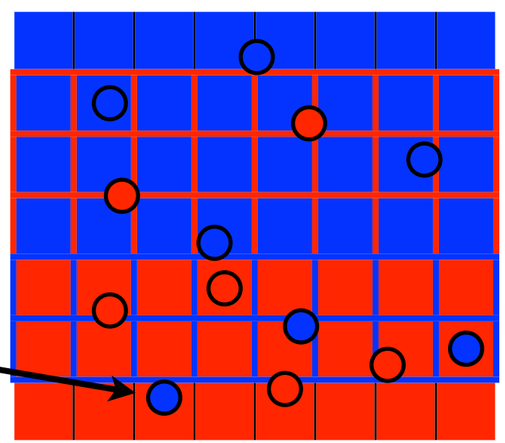
- Particles moving out of their GPUs partial box enter a 2-3 cell wide halo area
- The upper halo area is larger to accommodate lattice shift
- In border areas, intermediary results are exchanged between GPUs to calculate v_{cm}
- When a particle exits the halo area, particles are redistributed over the GPUs



Fluid particle handling in halo areas



- Particles moving out of their GPUs partial box enter a 2-3 cell wide halo area
- The upper halo area is larger to accommodate lattice shift
- In border areas, intermediary results are exchanged between GPUs to calculate v_{cm}
- When a particle exits the halo area, particles are redistributed over the GPUs



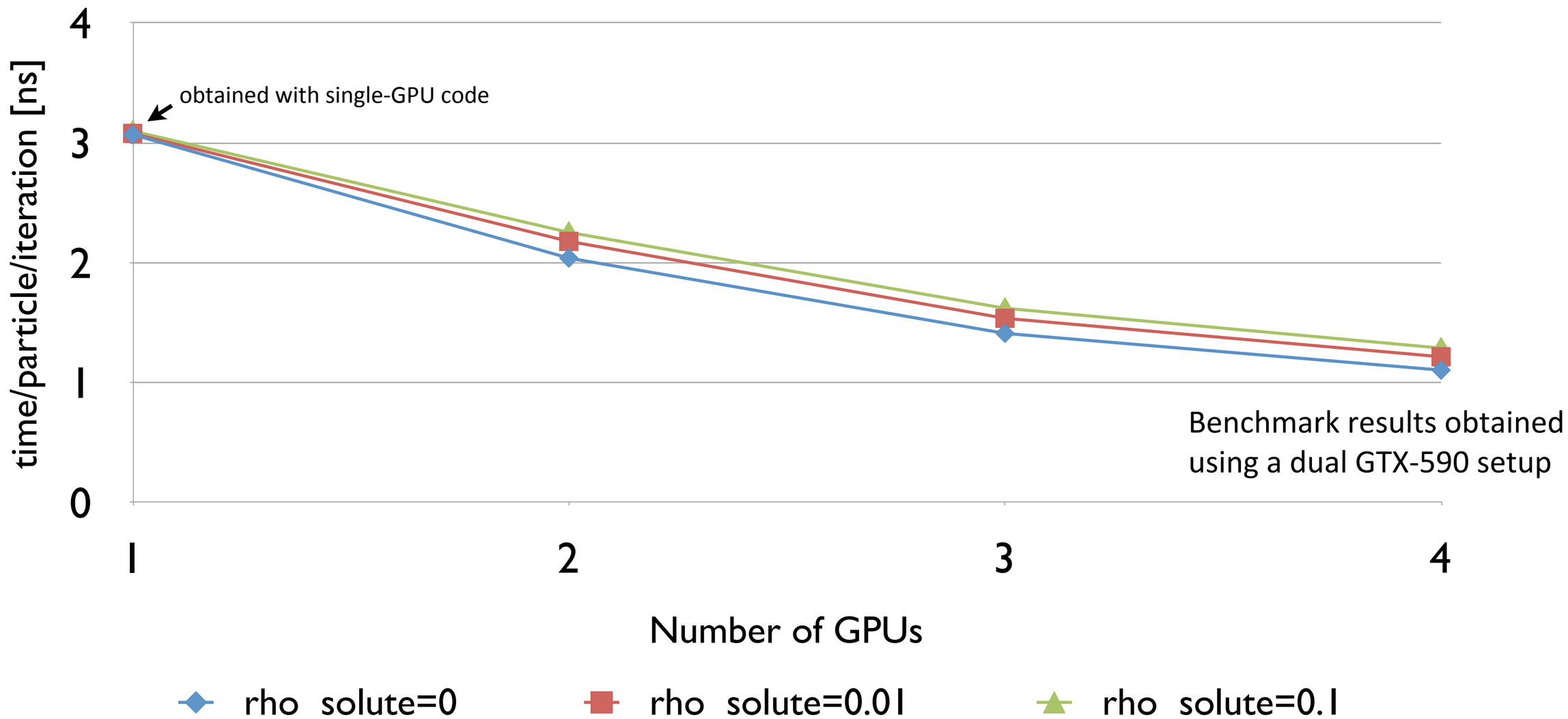
Solute particle handling in multi-GPU systems

- All solute particles are handled by GPU 0
- Their contribution to their cell's v_{cm} is calculated here
- Intermediary results are distributed to the appropriate GPU
- v_{cm} is retrieved to perform the rotation step

Multi-GPU optimizations

- Particles are inherently reordered for transfer between GPUs, so no explicit reordering necessary
- cudaMemcpyPeer blocks both involved GPUs
  - cudaMemcpyAsync is used parallel to calculations instead
    - using CUDA unified address space for GPUs on the same PCIe bus
    - using staging areas in pinned memory for quasi bidirectional transfer
- Caches and combined memory accesses are used as in the single GPU implementation

Multi-GPU benchmark results



Outlook

- Things to come:
  - Simplify domain decomposition for multi-GPU code
  - Merge single- and multi-GPU code
  - Different boundary conditions
  - MD functionality
  - Mapping of large simulation systems into smaller MPC-boxes

Acknowledgements

Special thanks to

Sunil Pratap Singh & Chien-Cheng Huang for providing serial code samples, testing and discussing results

Roland Winkler for initiating the project and providing answers to many, many questions...

Questions?