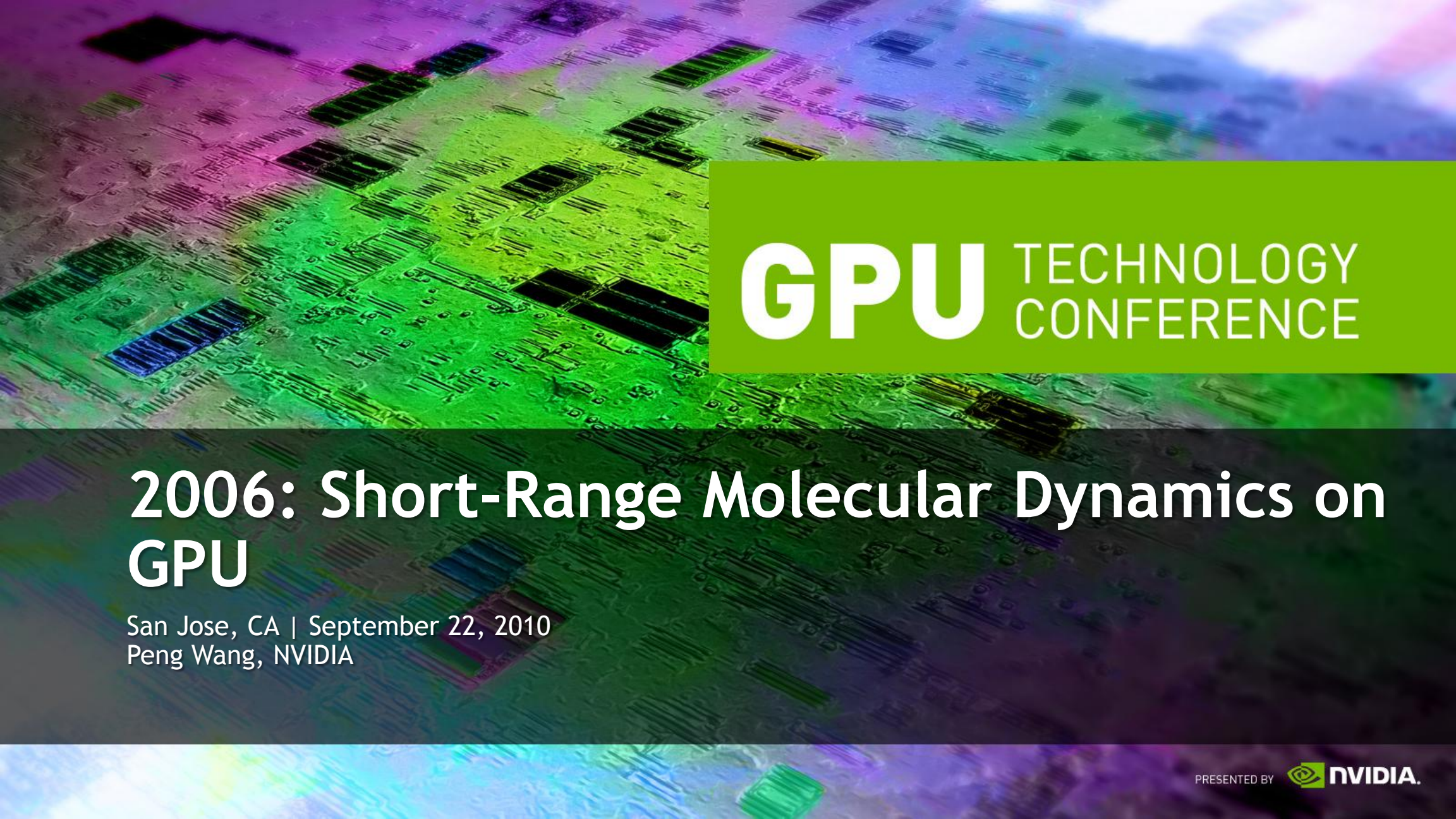


2006: Short-Range Molecular Dynamics on GPU

San Jose, CA | September 22, 2010

Peng Wang, NVIDIA

Overview

- The LAMMPS molecular dynamics (MD) code
- Cell-list generation and force calculation
  - Algorithm & performance analysis
- Neighbor-list generation and force calculation
  - Algorithm & performance analysis
- Neighbor-list data structure for special cases

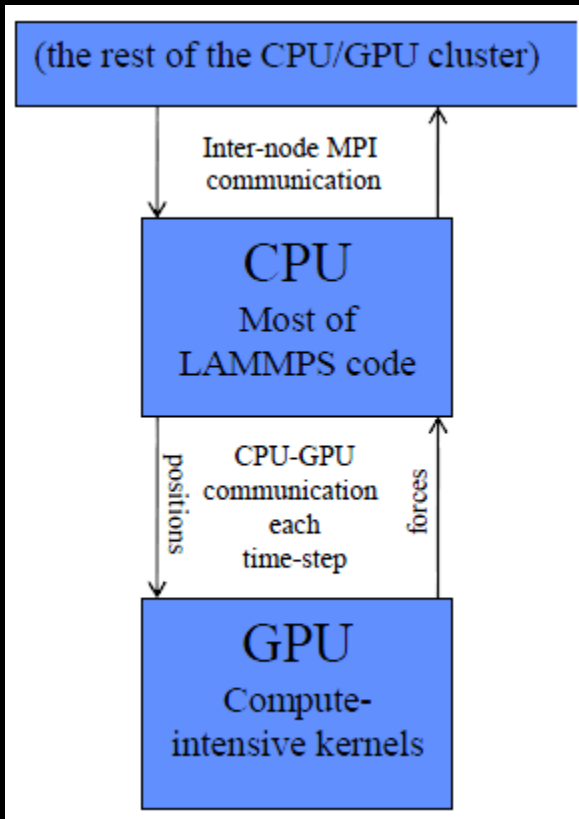
LAMMPS

http://lammps.sandia.gov

- Classical MD code in C++
  - Particles interact with neighbors within some cutoff
  - Gradient of the potential energy surface gives forces
  - Simulate by integrating equations of motion at timestep
- Open source, freely available for download under GPL
- MPI-parallelized using spatial-decomposition
- Atomistic, mesoscale, and coarse-grain simulations
- Variety of potentials (including many-body and coarse-grain)
- Variety of boundary conditions, constraints, etc

GPU-LAMMPS strategy

- Enable LAMMPS to run efficiently on GPU clusters.
- Not aiming for running on a single GPU.
- Not aiming to rewrite all of LAMMPS
- Rewrite only the most compute-intensive LAMMPS kernels in CUDA.
- At each time-step, ship particle positions from CPU to GPU, compute forces on the GPU, and then ship forces back to the CPU



MD algorithm

- N-body with a cutoff distance and a different force formula
- Naïve O(N^2) N-body like algorithm: not practical for large problems
- Avoiding searching all other particles
  - Cell-list (link-list)
  - Neighbor-list
- Arithmetic intensity depends on the force formula
  - Leonard-Jones among the least arithmetic intensity
  - Higher arithmetic intensity force will get better results

Leonard-Jones

- LJ CPU profile on a single Nehalem core
  - 108K atoms, ortho box (0, 50.4), lattice spacing 1.6796, cutoff=2.5, skin=0.3, neighbor_modify every 20, run=100
  - Total time: 10.85 sec.
  - Pair time (%) = 9.55339 (86.3939)
  - Neigh time (%) = 1.1004 (9.95123)
  - Comm time (%) = 0.0903673 (0.817216)
  - Outpt time (%) = 0.000515938 (0.00466577)
  - Other time (%) = 0.313272 (2.833)
- Force + Neighbor time is ~96% of total time

Some notations

- Two kinds of particles
  - Active particles: position needs update
  - Ghost particles: contribute to force
- Basic calculation:
  - Input: particle positions sorted in particle id
    - Id 0 to inum-1 is active particles
    - Id inum to nall-1 is ghost particles
  - Output: forces on each active particle

Cell-list

- Decompose the rectangular domain into cells
- Each cell has list of particles belonging to it
- cell length = cutoff distance + skin length
  - To avoid reconstruct the cell-list at every time step
  - Only need to reconstruct if some particle travel half of skin length
- When calculating force, every particle only needs to look at the 27 neighboring cells, saving the cost of looking at all other particles

GPU Data Structure for cell-list

- Most natural one: fixed size for each cell
- Advantage
  - Aligned access: the load address for the first thread in a warp is a multiple of the segment size
- Disadvantage
  - Another parameter that is hard to choose universally
  - Potentially a lot of wasted storage space, i.e. limit the problem size

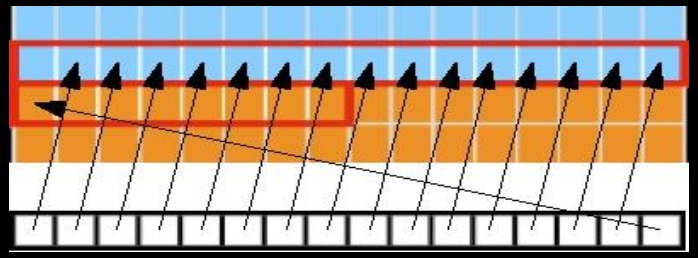
GPU Data Structure for cell-list

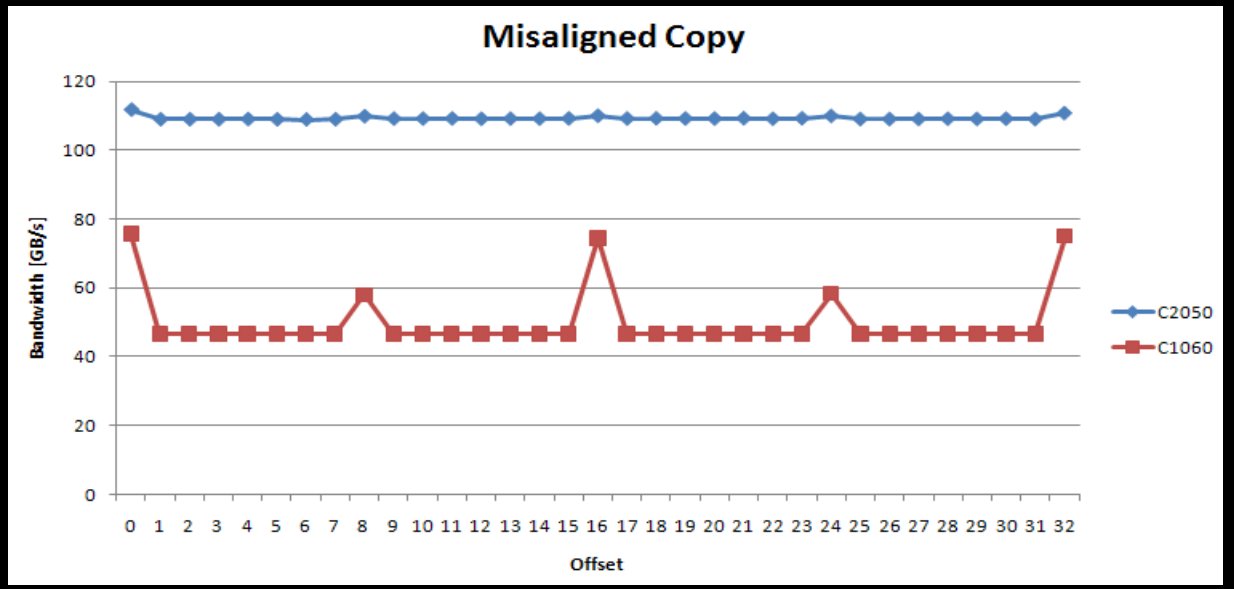
- Fixed bin size data structure not really necessary
  - Performance drop due to misaligned access is reduced by Fermi's new L1 cache
  - Force kernels using cell-list is not memory-bound, i.e. additional load time due to misaligned access is not overall important

Misaligned accesses on Tesla and Fermi

```
__global__ void offsetCopy(float *odata,
                           float* idata,
                           int offset)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
  odata[xid] = idata[xid];
}
```

offset=1





Data reuse among warps: L1 helps on misalgned access.

Packed GPU data-structure for cell-list

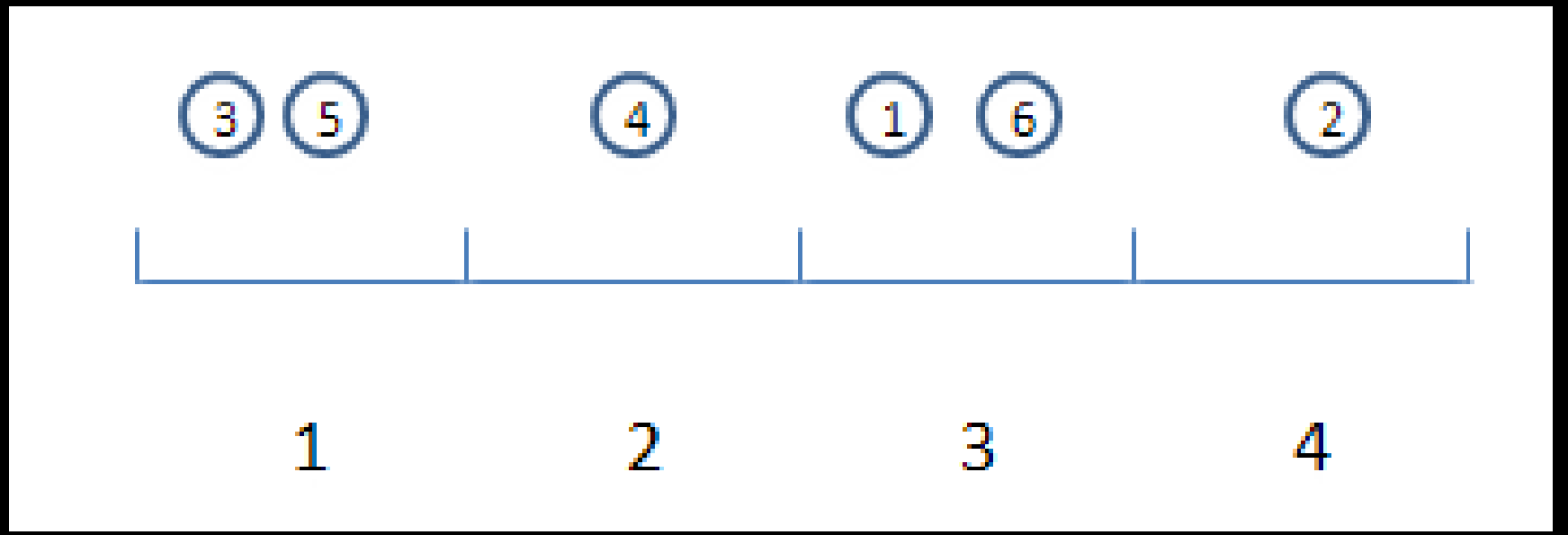
- Two arrays

```
typedef struct {
   int *cell_list;
   int *cell_counts;
} cell_list_gpu;
```

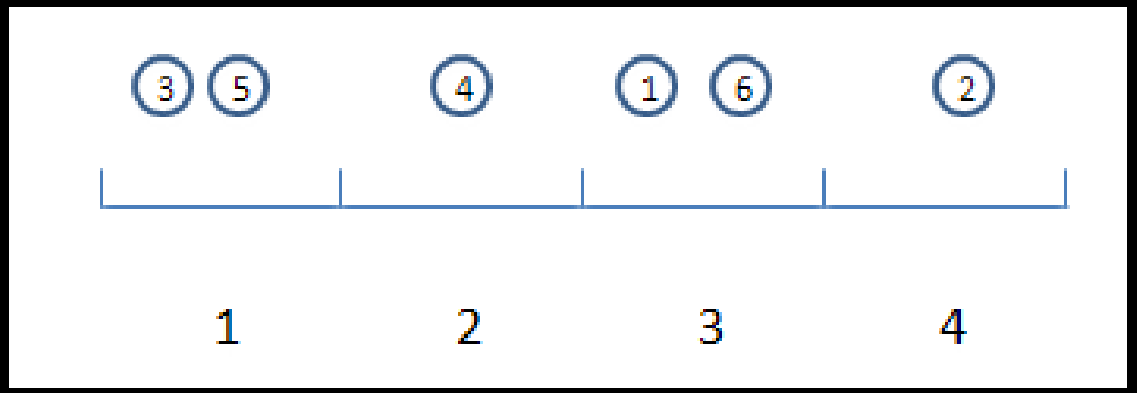
  - cell_list[nall]: list particle id belonging to each cell in a packed way
  - cell_counts[ncell+1]: list starting position of each cell in the cell_list array

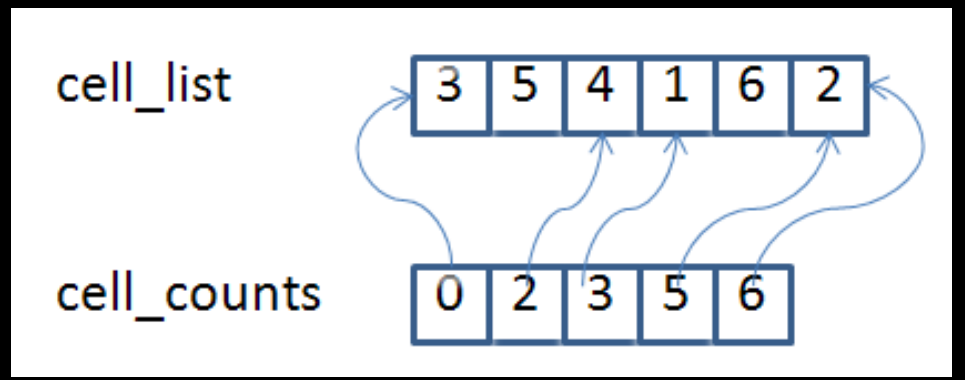
Cell-list build algorithm

- E.g.

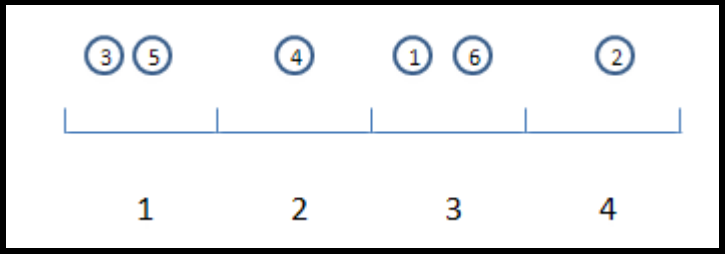


Cell-list build algorithm

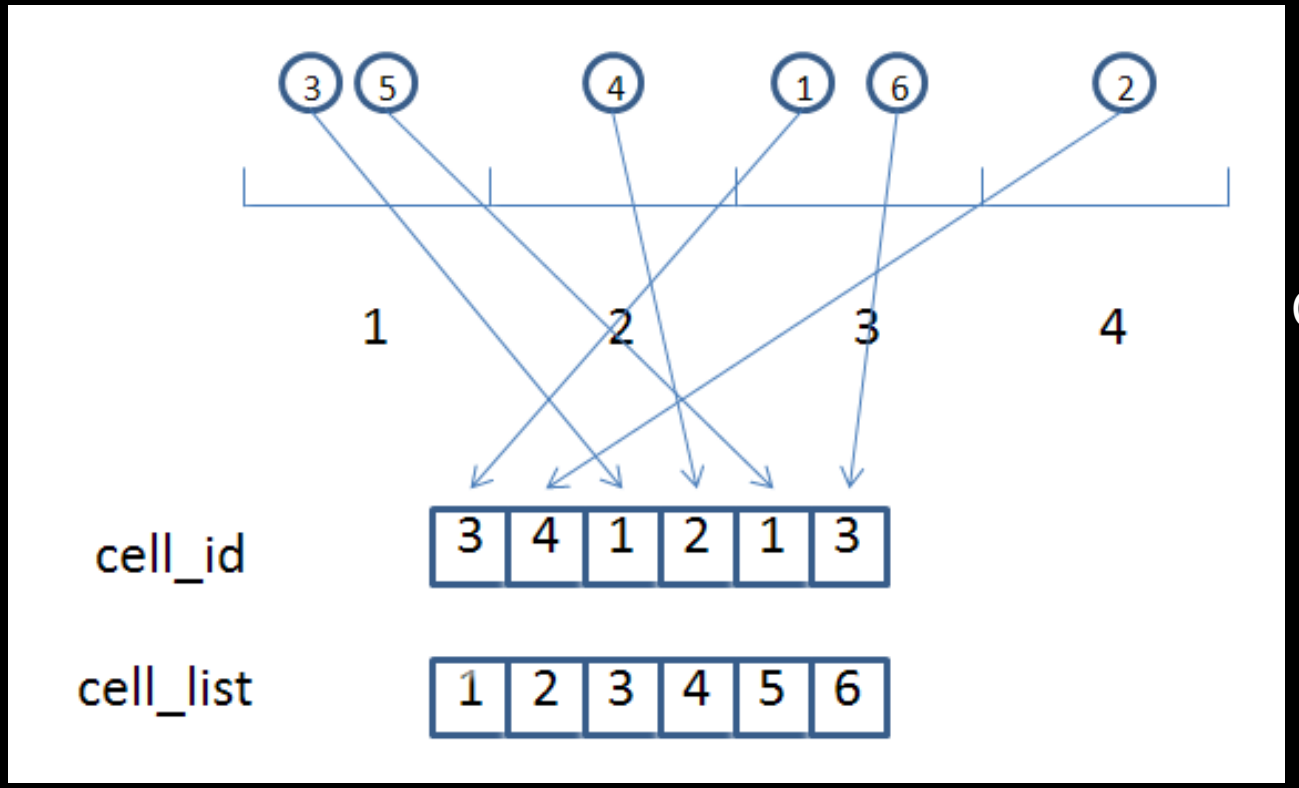




Cell-list build algorithm

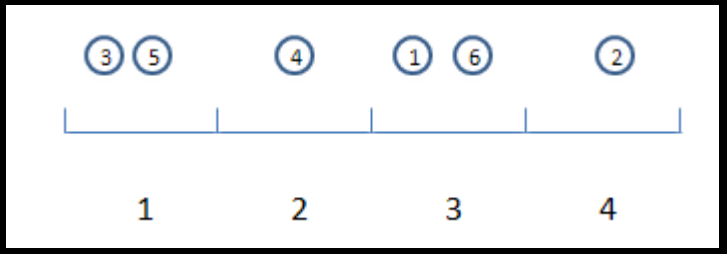


- Calculate the cell id of each particle, store the cell id to a temporary array cell_id, particle id to cell_list

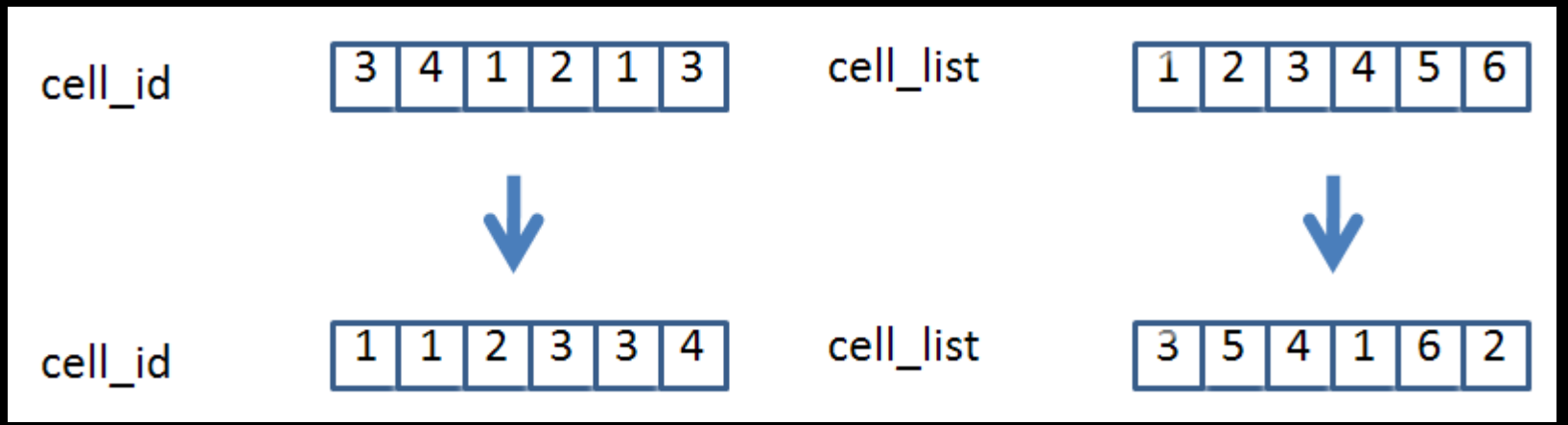


One thread per particle
Embarassingly parallel
Fully coalesced R/W

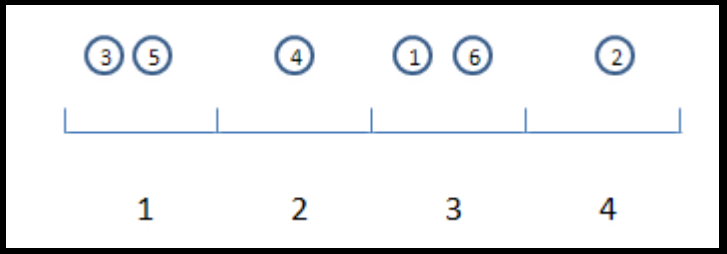
Cell-list build algorithm



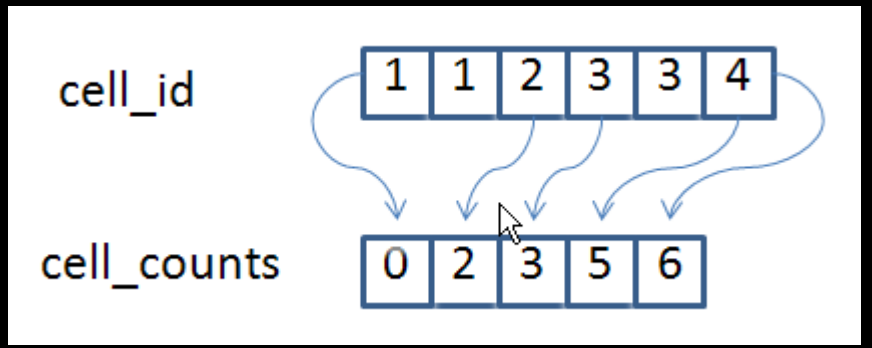
- Sort using cell_id as the key and cell_list as the value
  - Use CUDPP radix sort
  - cell_list then has the correct order



Cell-list build algorithm



- Calculate cell_counts from the sorted cell_id
  - One thread per particle
  - Compare cell id in the left and itself, if different, that's a cell boundary
  - Adding two special cases to handle the two boundaries



Force kernel using cell-list

- One CUDA block per cell, every thread calculating the force of one particle
- Loop through all the other particles in the 27 neighbors, if it's within the cutoff distance, accumulates its force contribution
- Data reuse: all threads are checking the same cell
  - Load the particle positions to shared memory

Force kernel with cell-list

```
Load position of current particle to P_i
for k in each 27 neighboring cell
    load positions of cell k to shared memory Ps
    for each particle j in Ps
        calculate r = dist(P_i, P_j)
        if r < cutoff distance && r > 1e-5
            calculate force from j to i
```


Cell-list performance

- C2050 performance
  - Total time: 2.9 sec (3.74x speedup over single Nehalem core)
  - Cell-list build time: 0.013 sec (~2.2 ms per cell-list build)
  - Force kernel time: 2.3 sec (~23 ms per force calculation)
- Conclusion:
  - For LJ force, C2050 using cell-list is comparable to 4 core Nehalem using neighbor-list
  - Cell-list build is ~10x cheaper than force kernel computation. So more frequent cell-list rebuild, even after every time step, is allowed (for better accuracy/using cell of half-cutoff distance)
  - CPU-GPU data transfer overhead not important

Kernel performance tuning mini guide

- Key questions in optimization:
  - What to optimize?
  - Are we done?
- Find out the limiting factor in kernel performance
  - Memory bandwidth/instruction throughput/latency bound
    - Rule-of-thumb: compare instruction-to-byte accessed to Fermi's peak instruction-issue-rate/bw~3.5.
    - Have good memory access pattern but effective memory throughput is low
    - Manually comment out computation and memory access: watch out for compiler tricks
- Measure effective memory/instruction throughput.
- Optimize for peak memory/instruction throughput

Force kernel performance

```
Load position of current particle to P_i
for k in each 27 neighboring cell
    load positions of cell k to shared memory Ps
    for each particle j in Ps
        calculate r = dist(P_i, P_j)
        if r < cutoff distance && r > 1e5
            calculate force from j to i
```

- Inside the search loop, comment out force calculation
  - By changing r2 > 1e-5 to r2 > 1e5
  - Time reduce from 22.5 ms to 20 ms
  - Force calculation is NOT the bottleneck (most of the time the conditional check fails)

Force kernel performance

```
Load position of current particle to P_i
for k in each 27 neighboring cell
    load positions of cell k to shared memory Ps
    for each particle j in Ps
        calculate r = dist(P_i, P_j)
        if r < cutoff distance && r > 1e-5
            calculate force from j to i
```

- Comment out the search on shared memory position but keep shared memory ld
  - By changing end_idx to end_idx/100
  - Time reduce from 22.5 ms to 6.5 ms
  - The kernel is large bound by instruction issue in the search loop
  - Roughly the search loop time is 22.5 - 6.5 ~ 15 ms

Force kernel performance

- There are 21 instructions inside the search loop when conditions fail
- On average 18.8 particles per cell, so each thread needs to check ~27*18.8~508 times
- Total threads=512K
- The search loop effective instruction throughput:
  - 21*508*512000/0.015*1e-9~377 Ginstr/s
  - ~73% of the peak instruction issue rate
  - We are close to done!

Why cell-list did not fly?

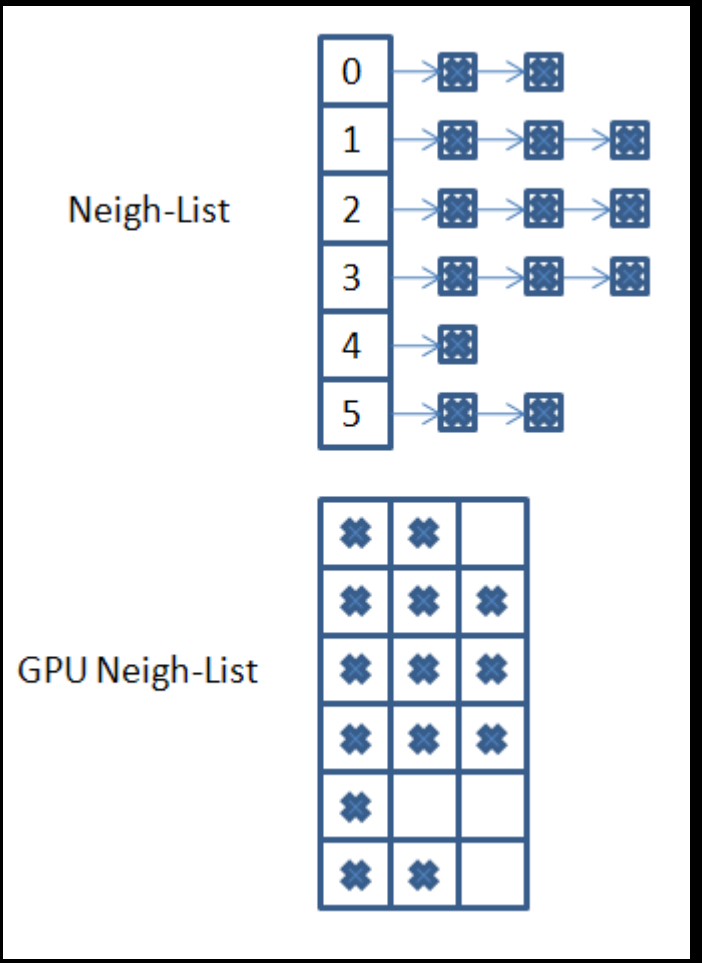
- For this initial condition, each particle has the same 54 neighbors which are within the cutoff distance and 78 neighbors which are within the cutoff + skin
- For cell-list, 54/508~11% of the search are successful, i.e. 90% of the instruction issued are “wasted”
- For neighbor-list, 54/78~70% of the search are successful
- So cell-list method just does much more “wasted” computation than the neighbor-list method

Neighbor-list

- Neighbor-list: stores the neighbors (particles within cutoff + skin) of each active particles
- Data structure requirement
  - Varying number of neighbors for different particles
  - Coalesced memory access

Neighbor-list data structure

- A regular 2D storage in column-majored way
- Advantage:
  - Simple for force kernel
  - Fully coalesced access (w/ padding)
- Disadvantage
  - Large waste of storage space for some problem
  - Limit the problem size



Neighbor-list build

- Unbinned algorithm: search all other particles O(N^2)
  - Not efficient for large problems
- Binned algorithm: first build a cell-list, then use that to build the neighbor-list
- We will use the binned algorithm for its efficiency on large problems

Neighbor-list build

- Build a cell-list as described in previous slides
- The neighbor-list build kernel is very similar to the force kernel using cell-list, where except of accumulating a force when a particle is within the cutoff distance, we add it to the neighbor-list

Force calculation using neighbor-list

- One thread per particle
- Each thread loops through the neighbor-list of its particle, accumulating to the force if a neighbor is within the cutoff distance

```
Load position of current particle to P_i
for each particle j in neighbor-list
    calculate r = dist(P_i, P_j)
    if r < cutoff distance && r > 1e-5
        calculate force from j to i
```

Neighbor-list performance

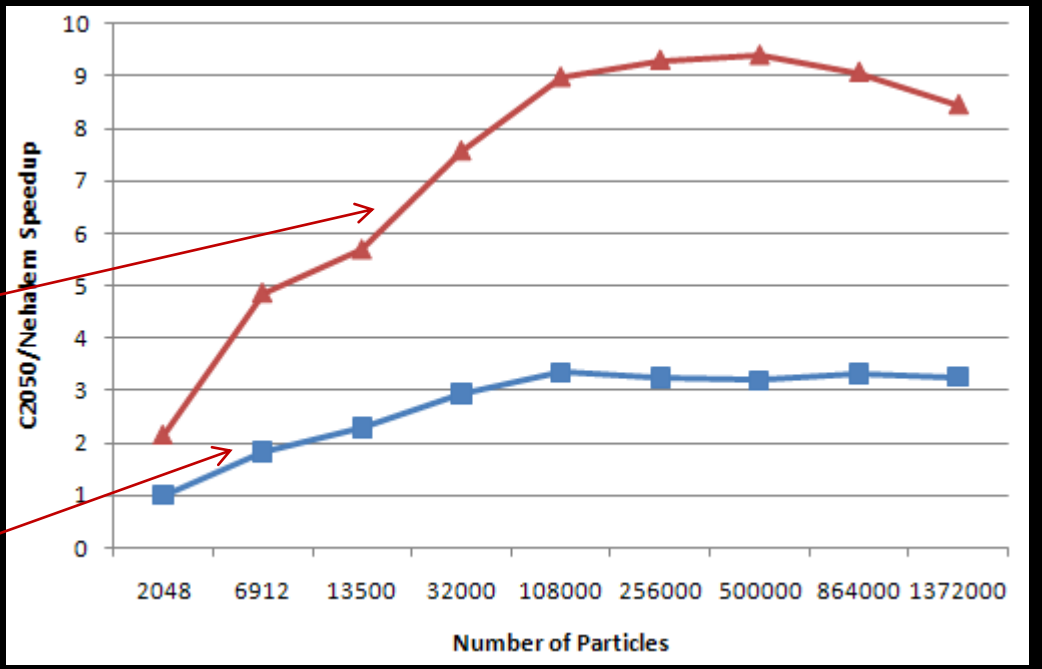
- C2050:
  - Total time: 1 sec (11x over a single Nehalem core)
  - Force time: 0.24 sec (2.4 ms per force calculation)
  - Neighbor-list build time: 0.16 sec (26 ms per build)
- Conclusion
  - Neighbor-list build is now ~10x more expensive than force calculation
  - Force + Neighbor is 35x over a single Nehalem core
  - Overhead important now

C2050 over Nehalem speedup

Speedup of a C2050 over 4 core Nehalem

Force + Neighbor list

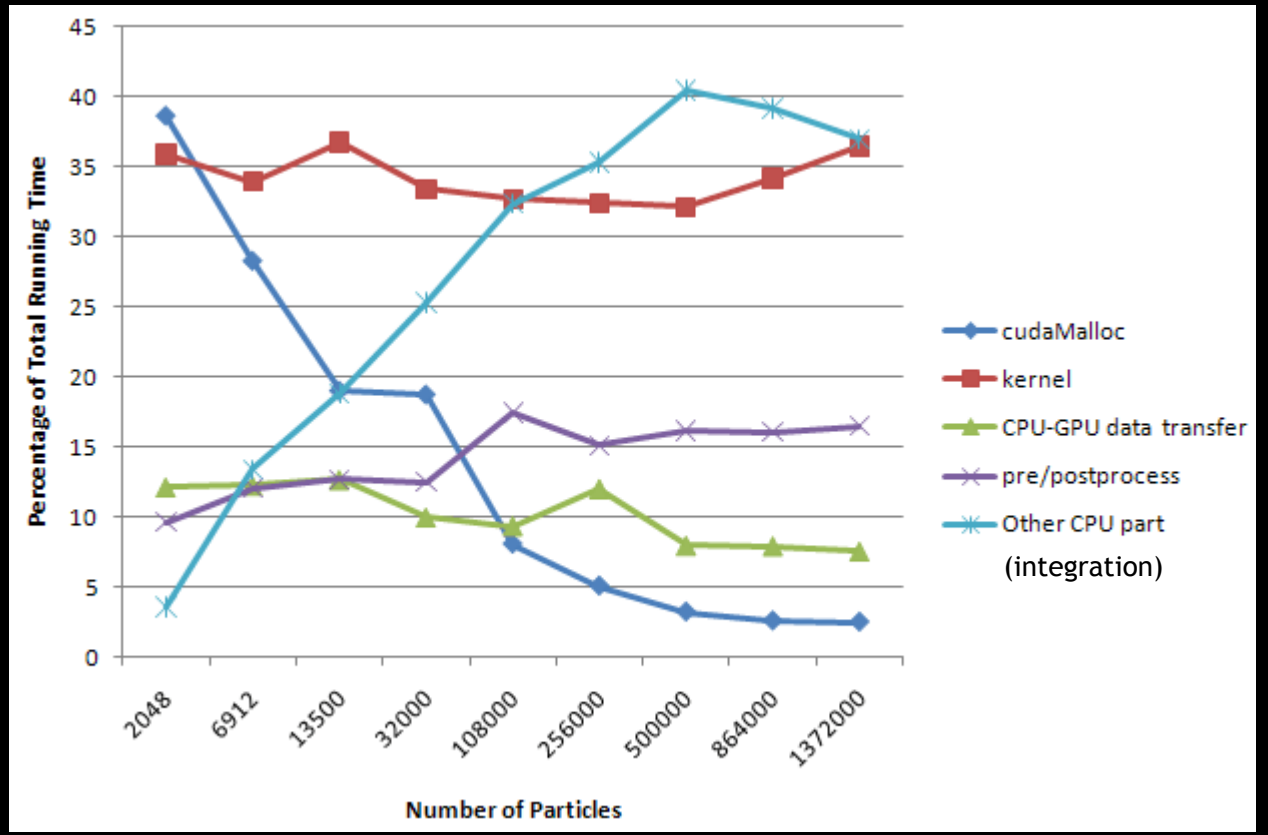
Overall speedup



Number of Particles

- LJ speedup will be the least significant.
- Overall speedup limited by Amdahl's law.

Profile



- cudaMalloc important at small np: user-managed memory
- Perf limited by time integration at large np: move that to GPU
- Only copy data back to CPU when MPI communication is required

Force kernel performance

- In the main loop
  - Within cutoff:
    - Instruction: 21
    - Memory load: 24
    - Ratio: 0.875
  - Outside cutoff:
    - Instruction: 8
    - Memory load: 24
    - Ratio: 0.33
- Neighbor-list force kernel is memory-bound

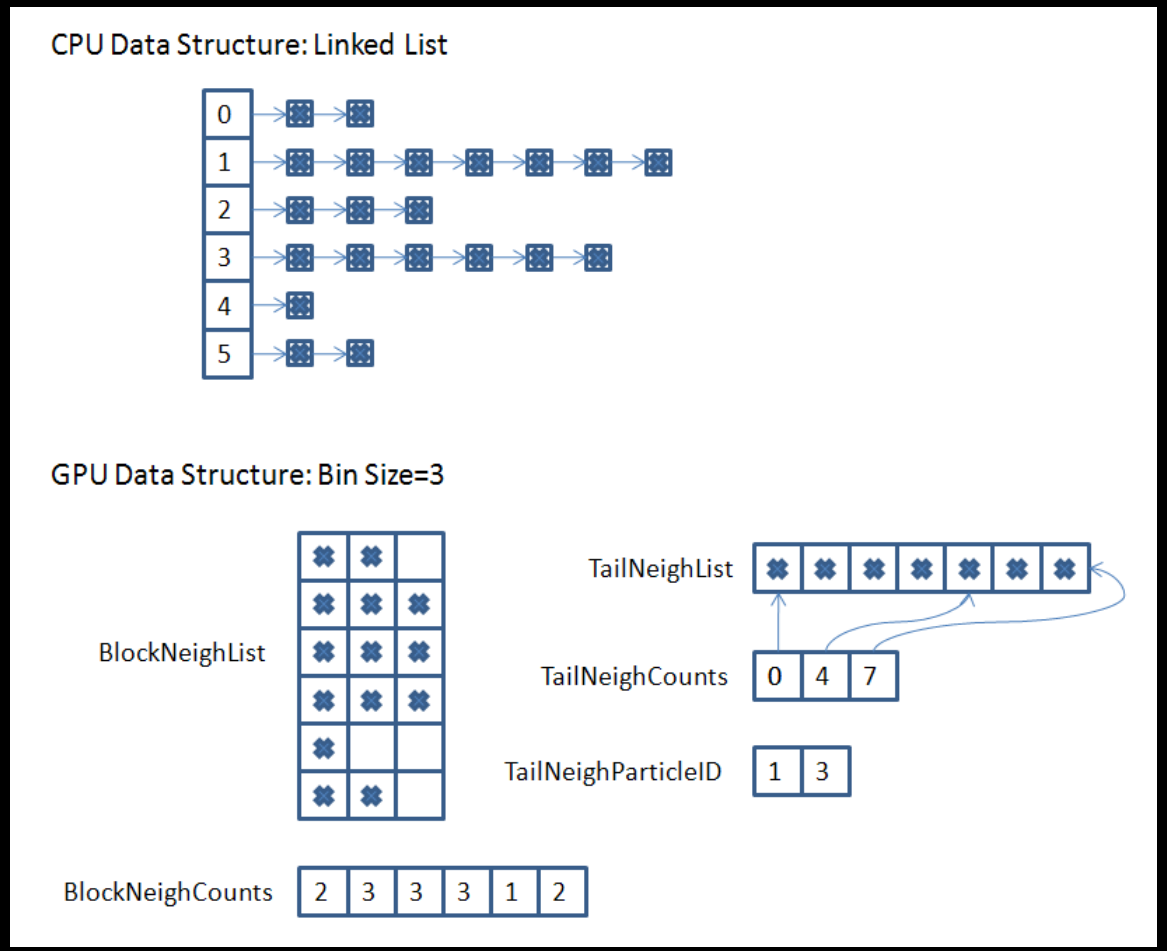
Force kernel performance

- Effective bandwidth: 127.8 GB/s
  - 84% of peak memory bandwidth
  - We are close to be done.
- Flop rate: 90.5 Gflop/s
- Texture is important
  - Turn off texture for position, we get only 43.1 GB/s

Tail neighbor-list

- Disadvantage of the NL data structure we are using so far
  - Large waste of storage space for some problem
  - Limit the problem size
- Idea: a hybrid data structure
  - Use the padded storage for typical cases (block NL)
  - Use a packed storage for exceptional cases (tail NL)
- Balance between performance and storage space
  - Most (>90%) of the entries in block NL

Tail neighbor-list

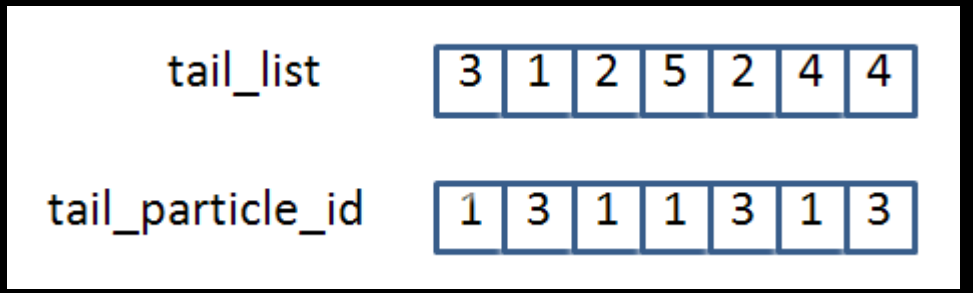


Summary

- Short-range MD algorithms can map well to GPU
- Cell-list and neighbor-list have difference performance characteristics
- Redesign data structure to optimize neighbor-list

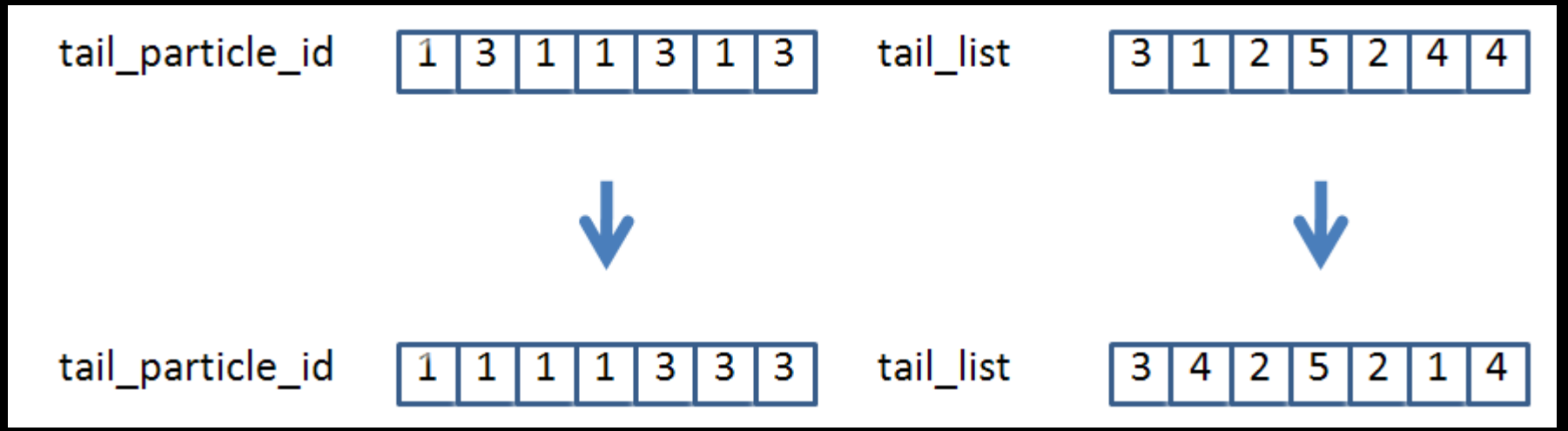
Build tail NL I

- In the NL build kernel, if neighbors > block NL size append new neighbor id and the corresponding particle id to two arrays atomically
- Because of using atomic functions, the tail NL is in a random unsorted state



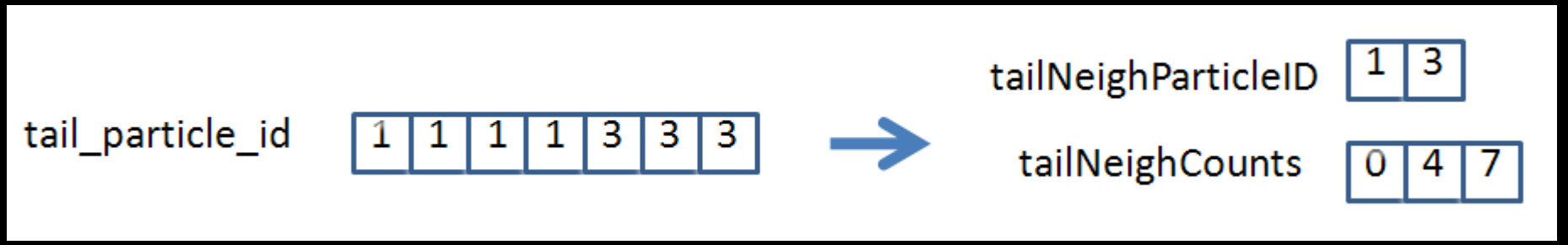
Build tail NL II

- Sort with tail_particle_id as key and tail_list as value



Build tail NL III

- Calculate tailNeighParticleID, tailNeighCounts from tail_particle_id
  - Stream compaction



Force kernel using tail NL

- One thread per particle
  - Simple to implement
  - Uncoalesced access
- One warp per particle
  - Coalesced access
  - Use shared memory for reduction
  - Waste of thread if # of neighbors is < warp size