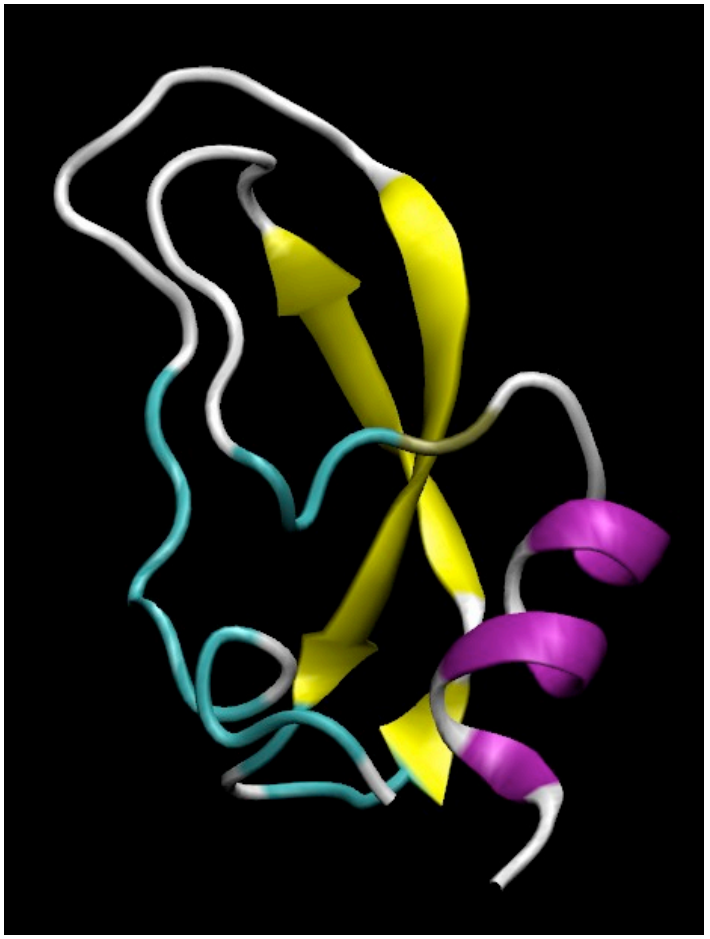
Molecular Dynamics Simulations on GPUs

Rossen Apostolov, Szilard Pall
Center for Biomembrane Research
Stockholm University

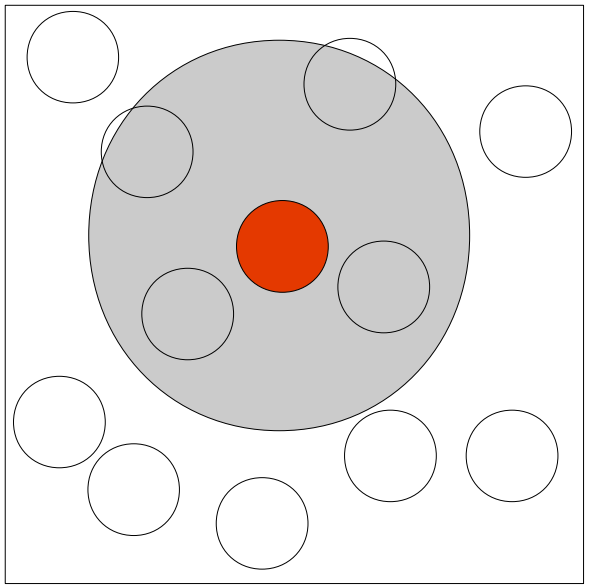
MD simulations on GPUs



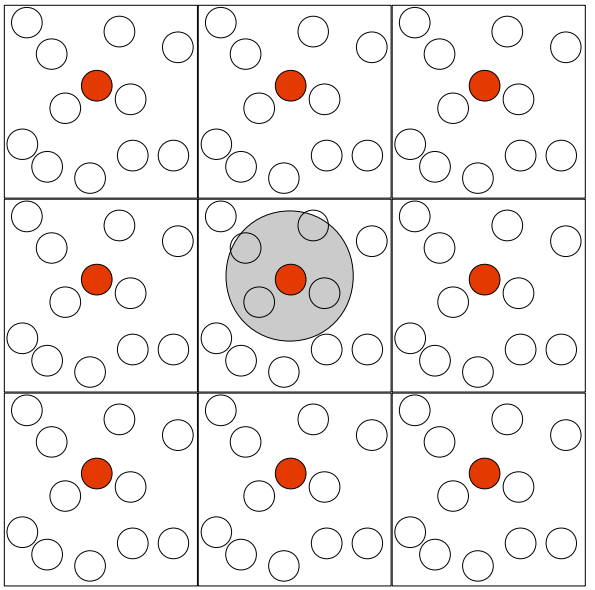
- Algorithms need to be reconsidered from the ground up
- Fast on CPU != Fast on GPU

  Slow on CPU != Slow on GPU

nonbonded interactions in molecular systems



simulated cell



same cell in
periodic boundary conditions

- Divide into short-range and long-range interactions
- Calculate short-range analytically
- Use approximations for the long-range

long-range interactions

- Van der Waals Potential

$$V_{LJ}(r_{ij}) = \frac{C_{ij}^{(12)}}{r_{ij}^{12}} - \frac{C_{ij}^{(6)}}{r_{ij}^{6}}$$

- Coulomb Potential

$$V_c(r_{ij}) = f\frac{q_i q_j}{\varepsilon_r r_{ij}}$$

outline

- short-range interactions
  - $O(N^2)$ (all-vs-all) algorithm
  - O(N) algorithm
- long-range electrostatics
  - Ewald summation $O(N^{3/2})$
  - Particle-Mesh Ewald (PME) O(N*logN)

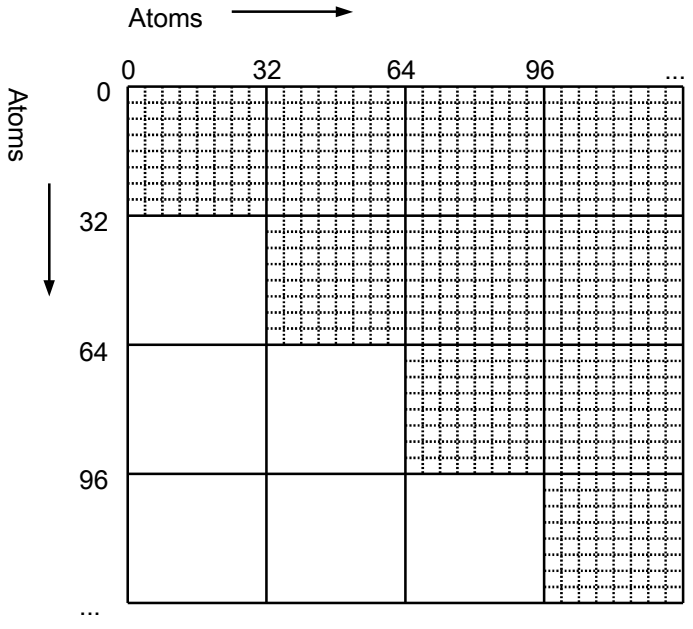
$O(N^2)$ (all-vs-all) algorithm on GPUs

efficient implementation of $O(N^2)$ algorithm would...

- minimize global memory access
- avoid thread synchronization
- take advantage of symmetry

algorithm

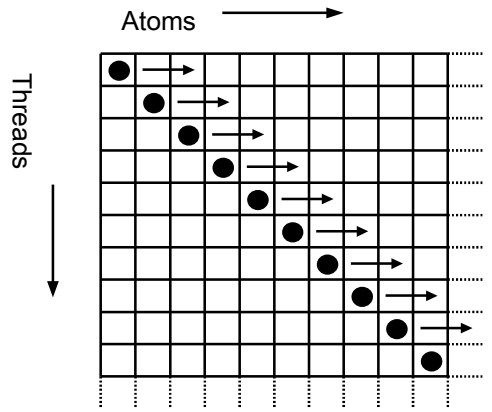
- Group atoms into blocks of 32
- Interactions divide into 32x32 tiles
- Each tile is processed by a group of 32 threads
  - Load atom coordinates and parameters into shared memory
  - Each thread computes interactions of one atom with 32 atoms
  - Use symmetry to skip half the tiles



Friedrichs et al., J Comput Chem 30: 864–872, 2009

algorithm (cont.)

- Each thread loops over atoms in a different order
  - Avoids conflicts between threads
- No explicit synchronization needed
  - Threads in a warp are always synchronized



O(N) algorithm on GPUs

why this is hard

- Traditional O(N) methods (e.g. neighbor lists) are slow on GPUs
  - Out of order memory access

```
for i = 1 to numNeighbors
    load coordinates and parameters for neighbor[i]
    compute force
    store force for neighbor[i]
```

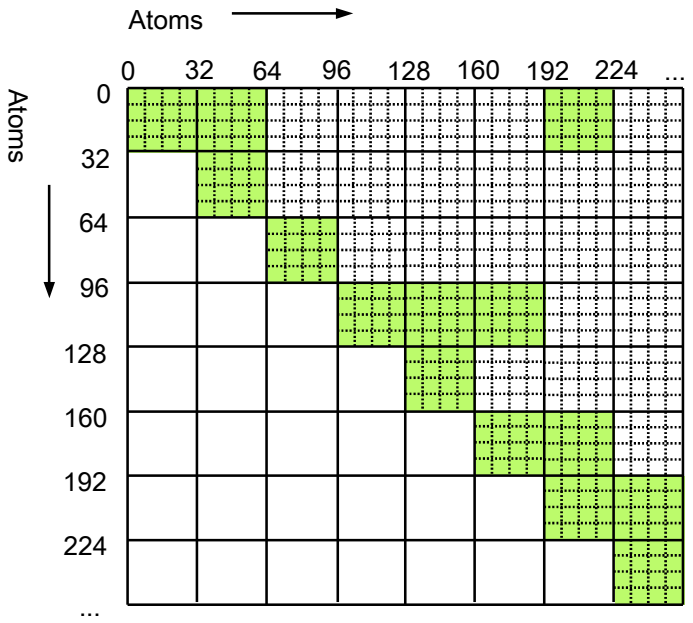
slow!

slow!

- The inner loop contains non-coalesced memory access!

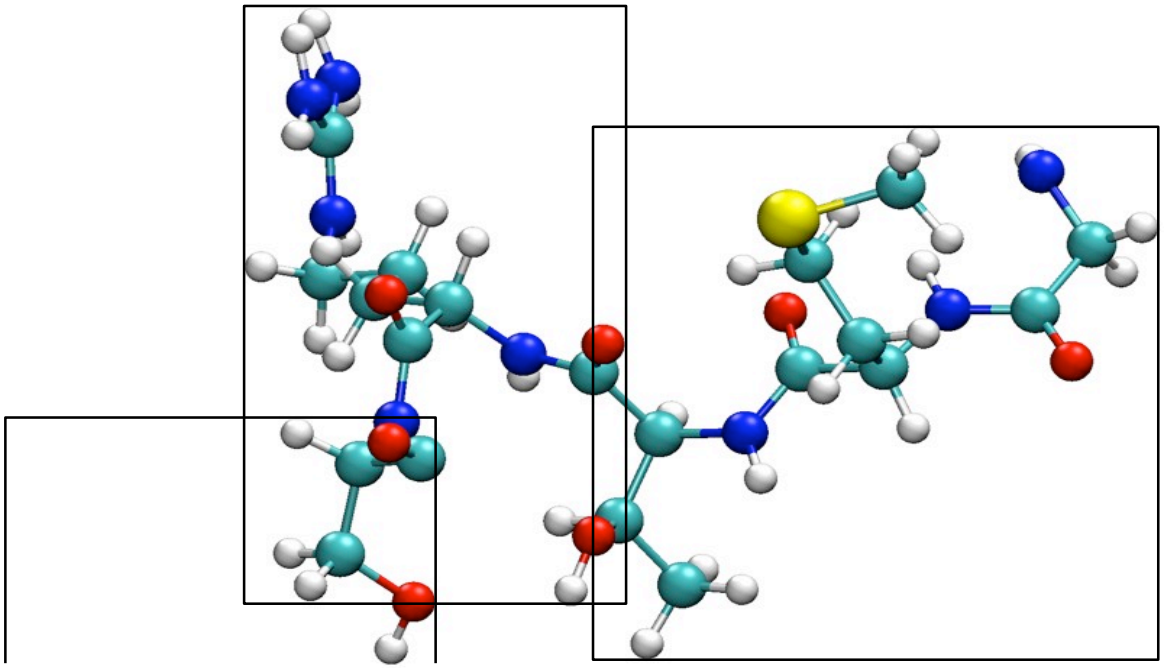
general approach

- Start with the $O(N^2)$ algorithm
- Exclude tiles with no interactions
  - Like a neighbor list between blocks of 32 atoms



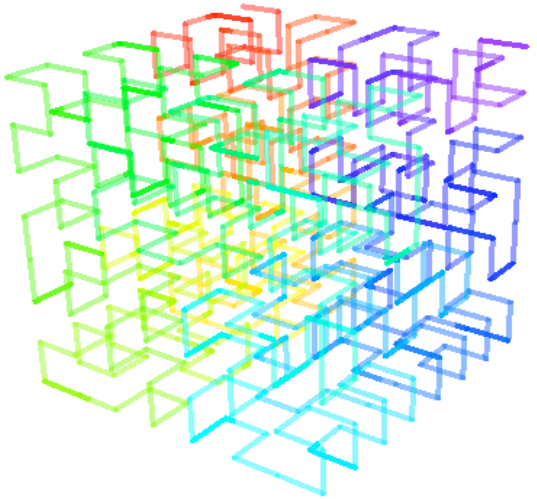
identifying tiles with interactions

- Compute an axis aligned bounding box for the 32 atoms in each block
- Calculate the distance between bounding boxes



solvent molecules

- Solvent molecules must be ordered to be spatially coherent
  - Otherwise bounding boxes will be very large
- Arrange along a space filling curve
- Reorder every ~100 time steps



performance of this algorithm

- Much faster than $O(N^2)$ for large systems
- Performance scales linearly

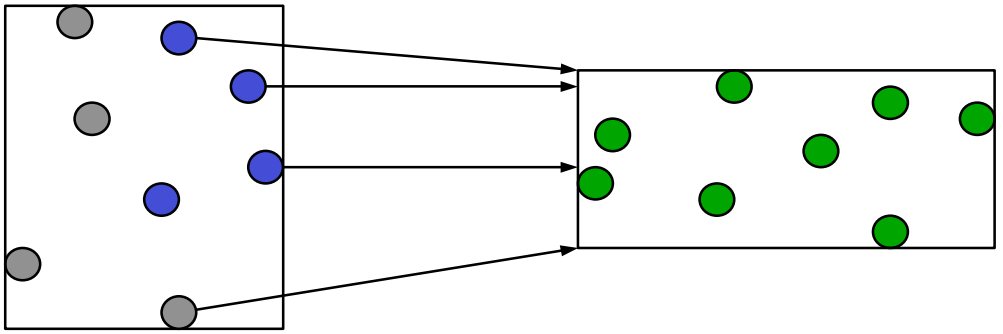
But

- Computes many more interactions than really required

  Computes all 1024 interactions in a tile, even if few (or none!) are within the cutoff

finer grained neighbor list

- For each tile with interactions:
  - Compute distance of *each atom* in one block from the *bounding box* of the other block
  - Set a flag for each atom

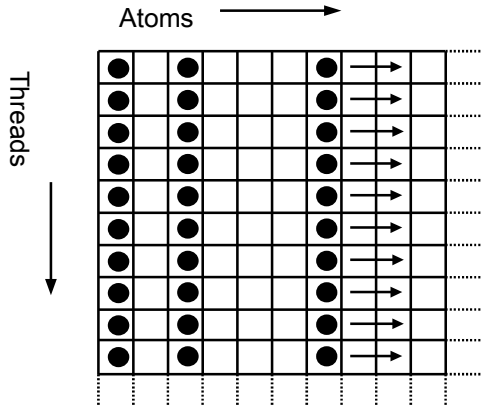


force computation for tile

```
for i = 1 to 32
  if (hasInteractions[i])
    compute interaction with atom i
```

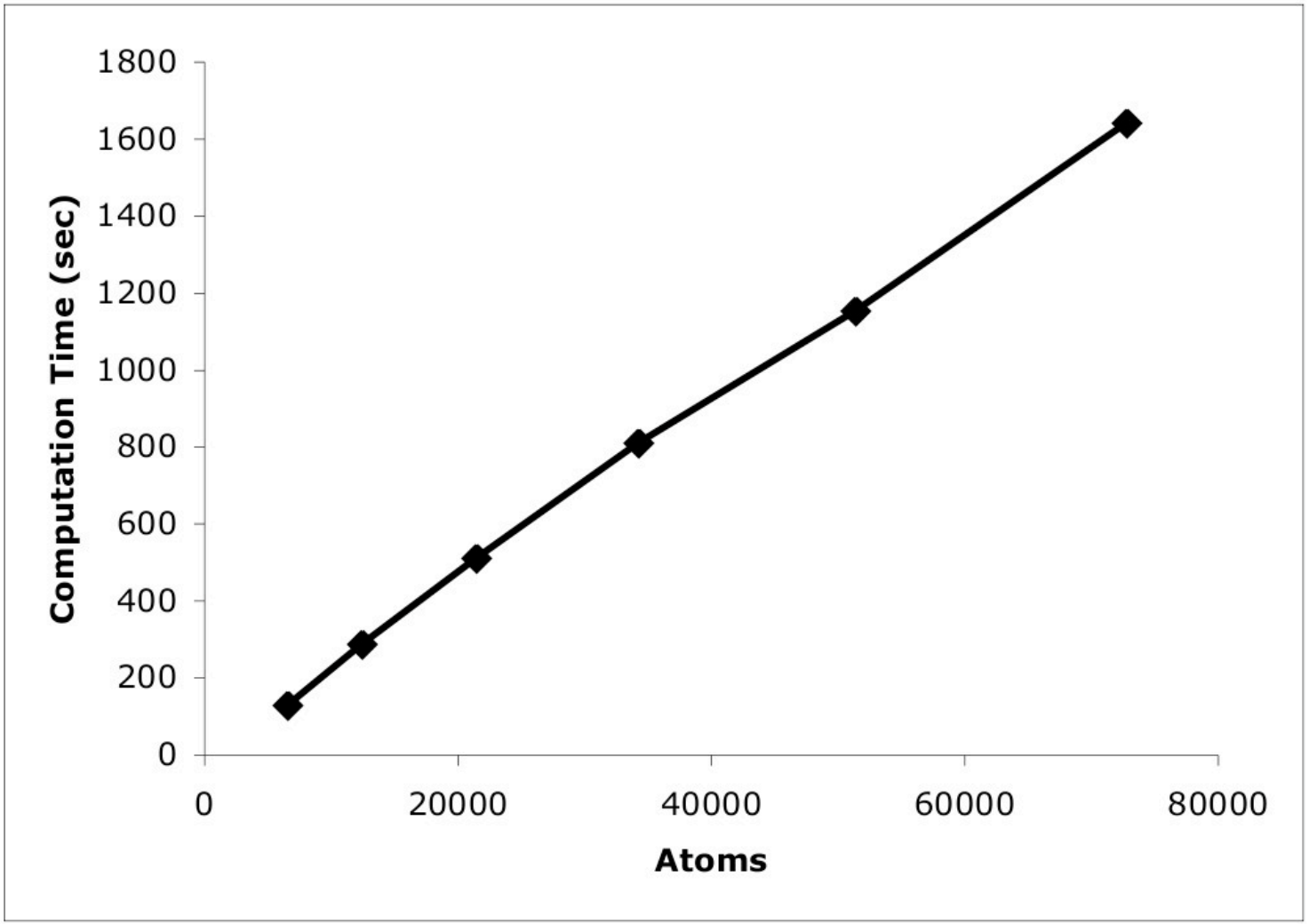
} on each thread

- All 32 threads must loop over atoms in the *same* order
  - Requires a reduction to sum the forces
  - For a few atoms, this is still much faster
  - For many atoms, better to just compute all interactions



Peter Eastmann et al., Efficient Nonbonded Interactions for Molecular Dynamics, J Comp Chem., 2010

benchmarks



Peter Eastmann et al., Efficient Nonbonded Interactions for Molecular Dynamics, J Comp Chem., 2010

benchmarks

proteins in explicit solvent

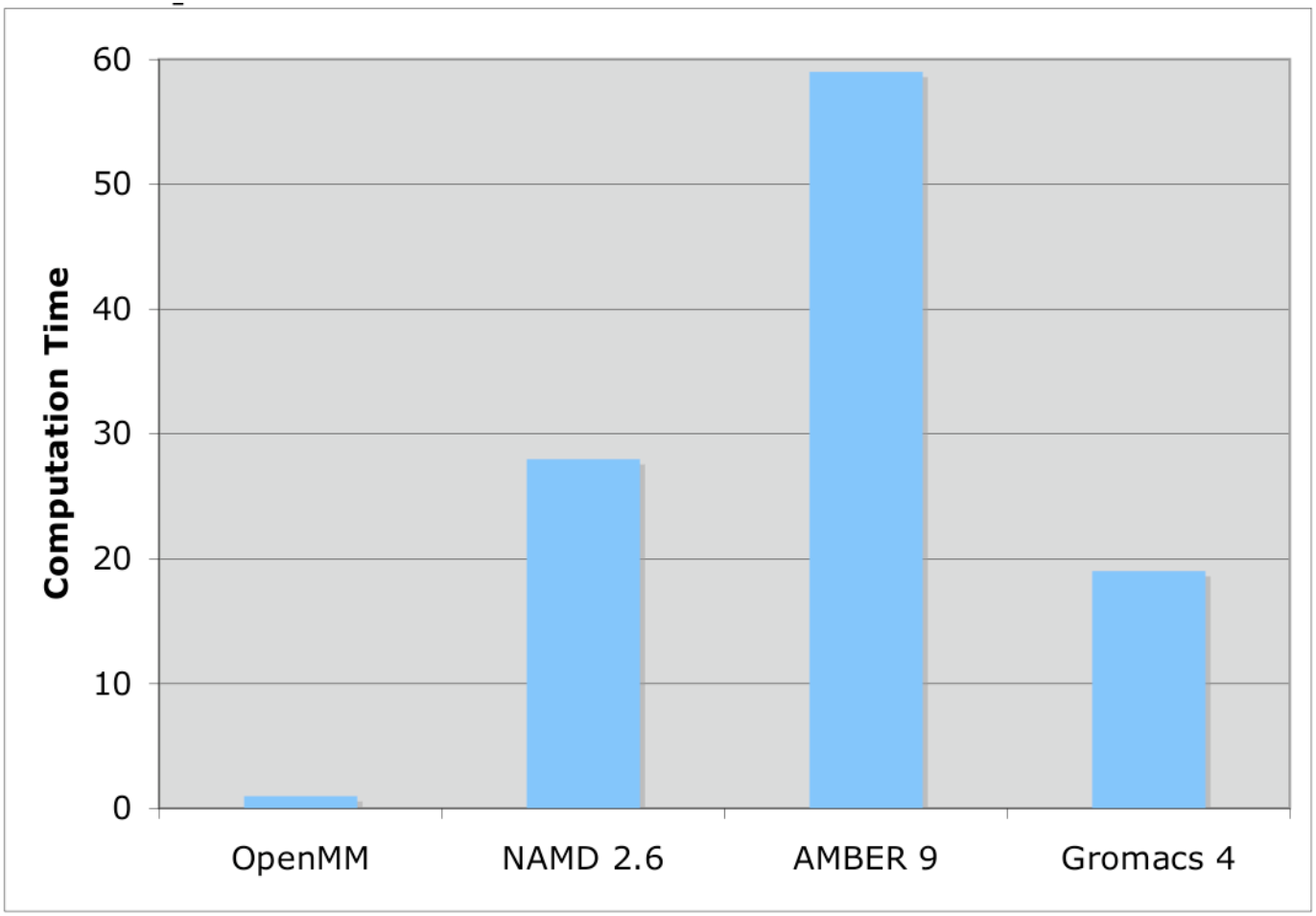
Protein	Total Atoms	Protein Atoms	ns/day
villin	8867	582	40
lambda	16437	1254	21
α-spectrin	73886	5078	5.0
fusion peptide in lipid bilayer	77373	834 (+26000 lipid atoms)	4.0

proteins in implicit solvent

Protein	Atoms	ns/day ($O(N^2)$ method)	ns/day (1 nm cutoff)
villin	582	528	402
lambda	1254	202	196
α-spectrin	5078	17	56

Peter Eastmann et al., Efficient Nonbonded Interactions for Molecular Dynamics, J Comp Chem., 2010

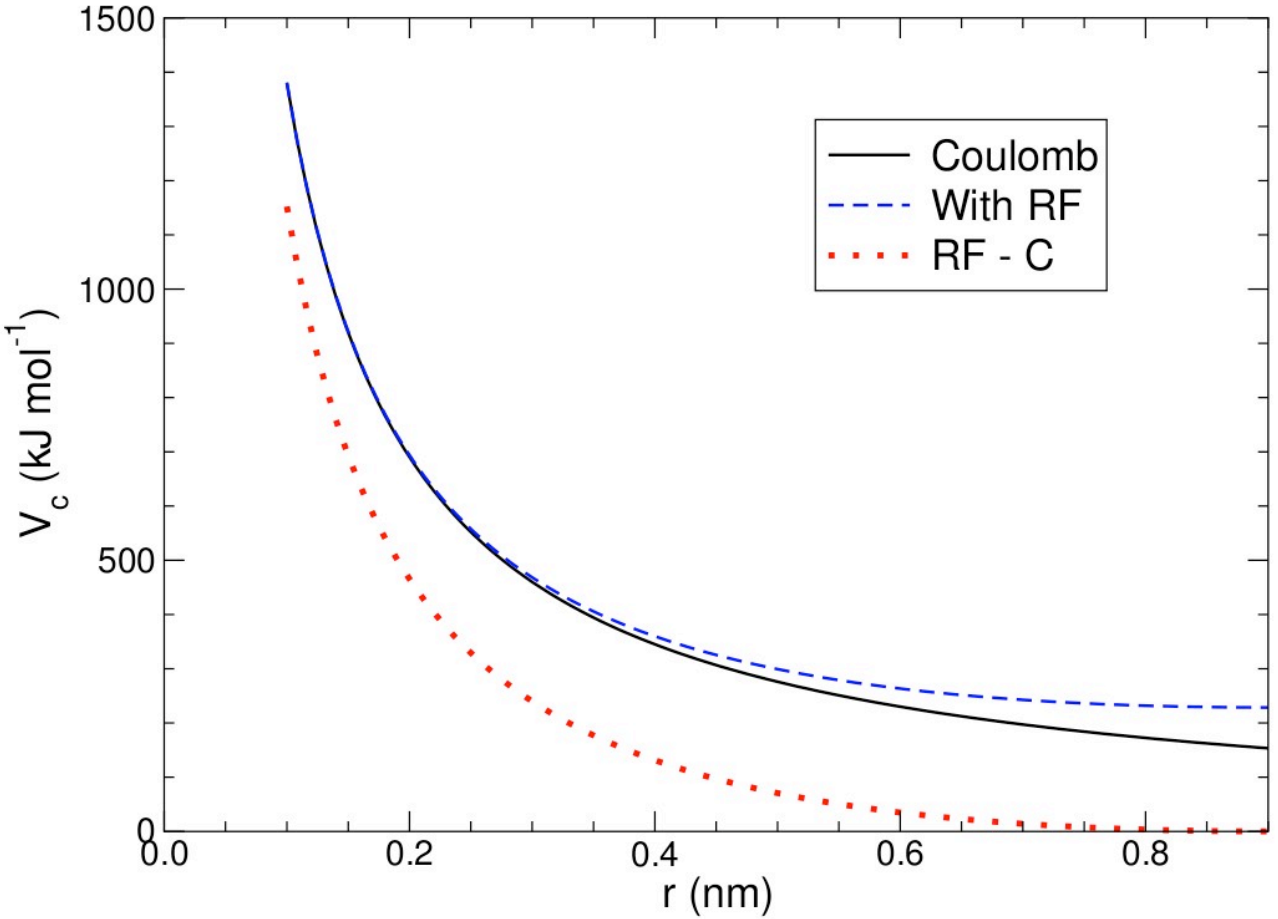
benchmarks



Single core computation time simulating lambda repressor
in explicit solvent (16437 atoms)
GPU: Nvidia GTX280
CPU: 3.0 GHz Intel Core 2 Duo

long-range electrostatics on GPUs

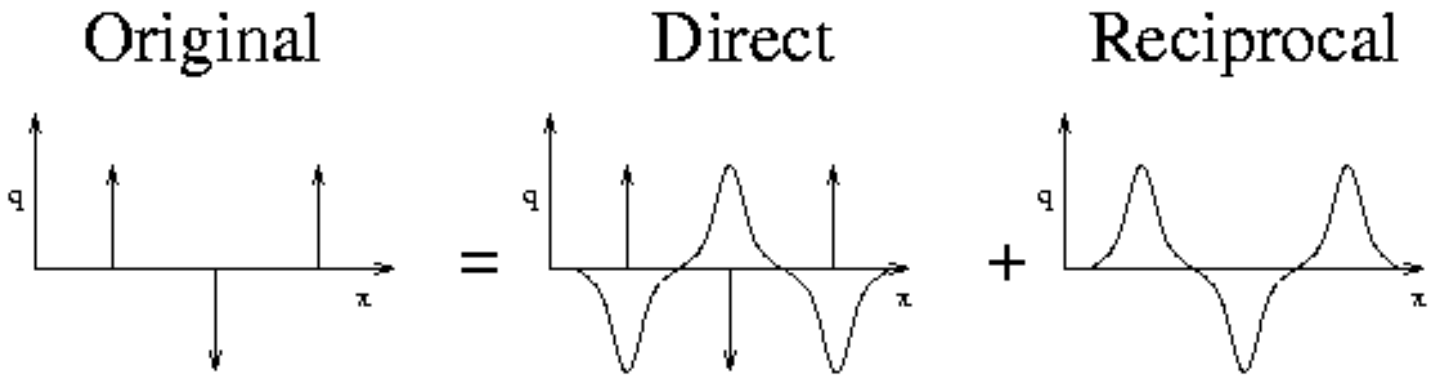
reaction-field method



$$V_{crf} = f \frac{q_i q_j}{\varepsilon_r} \left[\frac{1}{r_{ij}} + k_{rf}\, r_{ij}^2 - c_{rf} \right]$$

Ewald summation method

Ewald summation method
O(N^2) algorithm



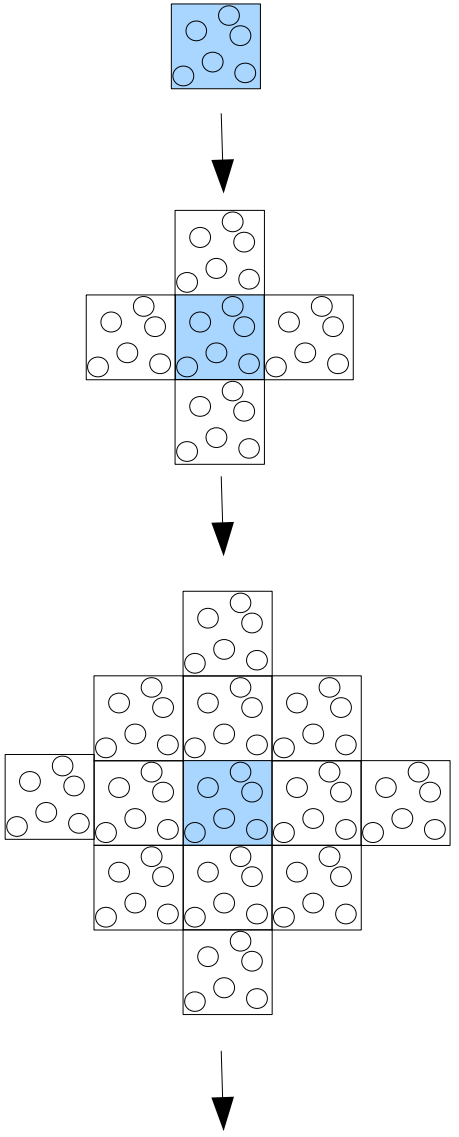
$$V = \frac{f}{2} \sum_{n_x} \sum_{n_y} \sum_{n_z *} \sum_{i}^{N} \sum_{j}^{N} \frac{q_i q_j}{\mathbf{r}_{ij,\mathbf{n}}}$$

$$V = V_{dir} + V_{rec} + V_0$$

$$V_{dir} = \frac{f}{2} \sum_{i,j}^{N} \sum_{n_x} \sum_{n_y} \sum_{n_z *} q_i q_j \frac{\operatorname{erfc}(\beta r_{ij,\mathbf{n}})}{r_{ij,\mathbf{n}}}$$

$$V_{rec} = \frac{f}{2\pi V} \sum_{i,j}^{N} q_i q_j \sum_{m_x} \sum_{m_y} \sum_{m_z *} \frac{\exp\left(-(\pi \mathbf{m}/\beta)^2 + 2\pi i \mathbf{m} \cdot (\mathbf{r}_i - \mathbf{r}_j)\right)}{\mathbf{m}^2}$$

$$V_0 = -\frac{f\beta}{\sqrt{\pi}} \sum_{i}^{N} q_i^2$$



Ewald summation method O(N^3/2) algorithm

$$\vec{F}_i(\vec{R}) = -\nabla U_c(\vec{R})$$

$$= \frac{1}{4\pi\varepsilon_0} \sum_n \sum_{\substack{j=1 \\ j\neq i}}^{N^*} q_i q_j \left\{ \frac{\operatorname{erfc}(\alpha|r_{ij}+n|)}{|r_{ij}+n|} + \frac{2\alpha}{\sqrt{\pi}} \mathrm{e}^{-\alpha^2|r_{ij}+n|^2} \right\} \frac{r_{ij}+n}{|r_{ij}+n|^2}$$

$$+ \frac{2}{\varepsilon_0 V} \sum_{\mathbf{k}>0} \frac{q_i \mathbf{k}}{k^2} \mathrm{e}^{-\frac{k^2}{4\alpha^2}} \left\{ \sin(\mathbf{k}\cdot r_i) \sum_{j=1}^{N} q_j \cos(\mathbf{k}\cdot r_j) + \cos(\mathbf{k}\cdot r_i) \sum_{j=1}^{N} q_j \sin(\mathbf{k}\cdot r_j) \right\}$$

$$- \frac{1}{4\pi\varepsilon_0} \sum_{i\leqslant j}^{M^*} \frac{q_i q_j}{r_{ij}^3} \left\{ \operatorname{erf}(\alpha r_{ij}) - \frac{2\alpha r_{ij}}{\sqrt{\pi}} \mathrm{e}^{-\alpha^2|r_{ij}+n|^2} \right\}$$

```
for each wave vector K

    CosSum = 0
    SinSum = 0

    for each atom i
        StructureFactor_Real(i) = charge(i) * cos(K.coord(i))
        StructureFactor_Imag(i) = charge(i) * sin(K.coord(i))

        CosSum += StructureFactor_Real(i)
        SinSum += StructureFactor_Imag(i)

    for each atom i
        force(i) = CosSum * StructureFactor_Imag(i)
                 - SinSum * StructureFactor_Real(i)
```

Ewald $O(N^{3/2})$ algorithm on GPUs

CalculateCosSinSums_kernel()

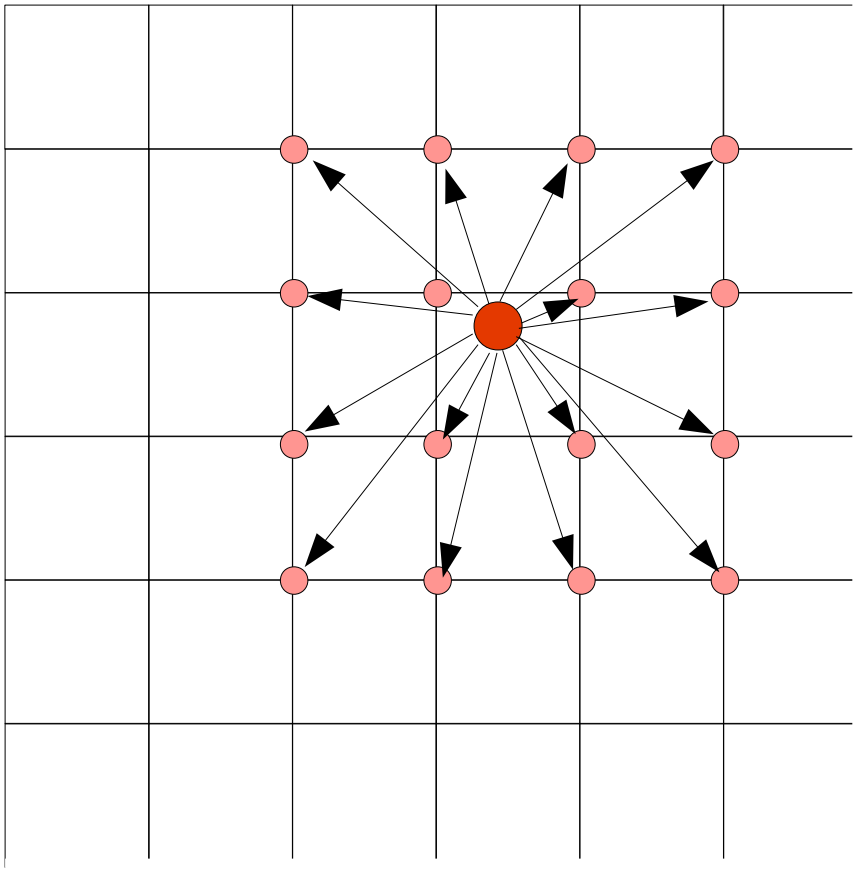
```
// Each thread is processing
// one of the wave vectors K

K = threadIdx.x + blockIdx.x * blockDim.x;

// Compute the sum for that K

float2 sum = make_float2(0.0f, 0.0f);

// Read atom data
for each atom i
    sum.x += charge(i) * cos(K.coord(i))
    sum.y += charge(i) * sin(K.coord(i))

// Store the sum in global memory
    Sum(K) = sum;

// Compute the contribution to the energy.
    energy += sum * sum

// Store energy in global memory
    Energy(K) = energy
```

CalculateEwaldForces_kernel()

```
// Each thread is processing
// one of the atoms

atom = threadIdx.x + blockIdx.x *
blockDim.x;

// Load atom data
 force = Force(atom)
 coord = Coord(atom)

// Loop over all wave vectors.
// Compute the force contribution of this
wave vector.

  for each wave vector K

// calculate again (!) the structure
// factor for this wave vector
   structureFactor.x = cos(K.coord(i))
   structureFactor.y = sin(K.coord(i))

// Load the Cosine/Sine Sum from memory
   cosSinSum = cSim.pEwaldCosSinSum[K];
   force += charge(i) *
      (cosSinSum.x*structureFactor.y-
       cosSinSum.y*StructureFactor.x)

// Record the force on the atom.
    Force(atom) = force
```

Particle-Mesh Ewald (PME) method

Particle-Mesh Ewald method



1) calculate scaled fractionals

2) calculate B-spline coefficient

3) “scatterr” the charges over the grid points from spline-coefficient

4) execute forward FFT

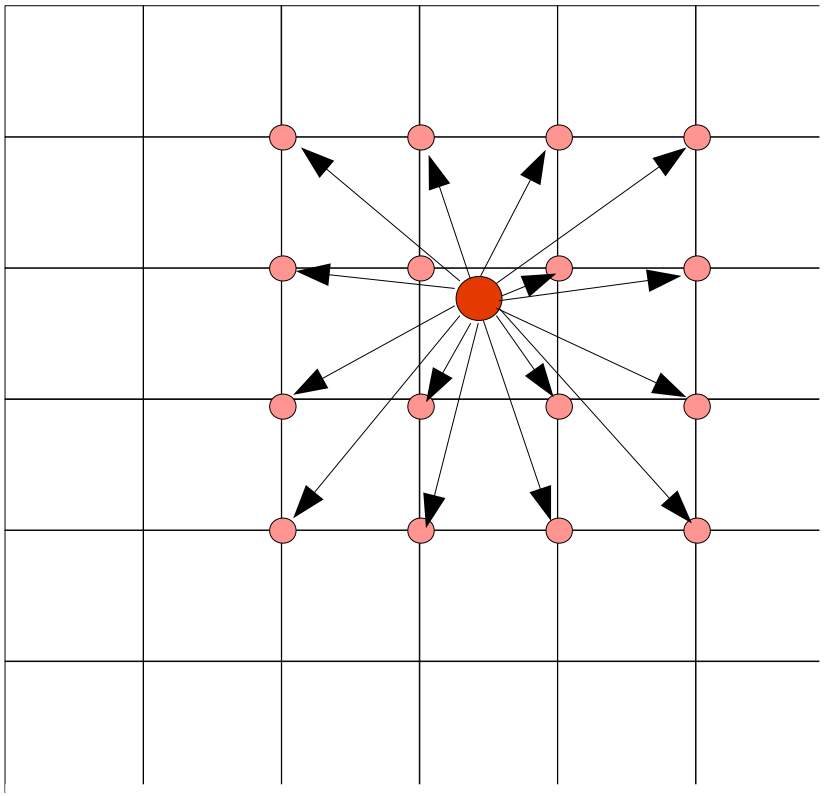
5) calculate the reciprocal energy

6) execute backward FFT

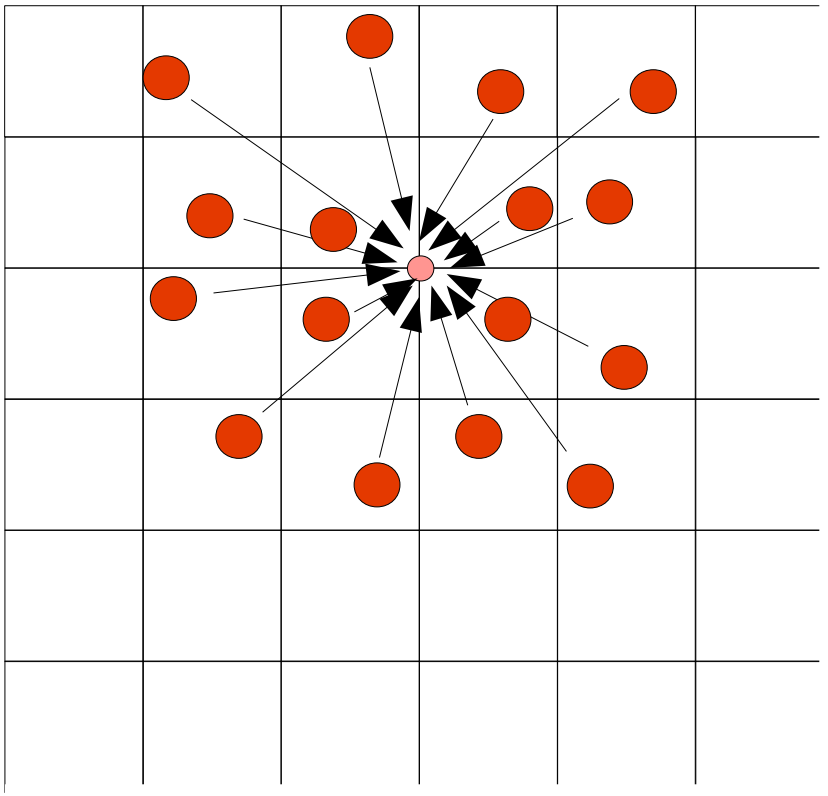
7) calculate force gradient

Particle-Mesh Ewald method

CPU
spreading of charges

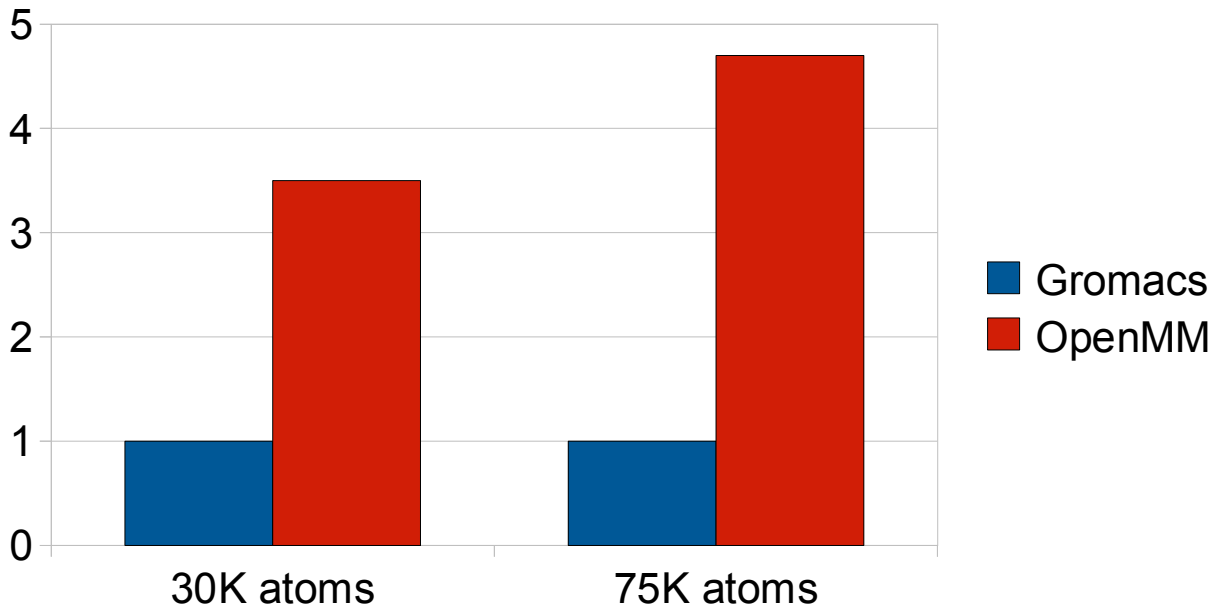


GPU
gathering of charges



- sort the atoms before gather !

benchmarks



single core computation time for solvated spectrin system
GPU: Nvidia GTX280
CPU: 3.0 GHz Intel Core 2 Duo

Acknowledgements

Vijay Pande (Stanford University)

Peter Eastman (Stanford University)

Scott Le Grand (Nvidia)

Mike Houston (AMD)

OpenMM – Outline

- OpenMM = Open Molecular Mechanics
  - Yet another MD package?
  - Why OpenMM?
  - Goals & API Design Principles
  - Current Capabilities
  - Summary
- OpenMM Architecture
  - Platform Dependent Code
  - Platform Independent Code
  - OpenMM in Use
- Gromacs-OpenMM
  - Integration Quirks
  - Availability

OpenMM = Open Molecular Mechanics

- OpenMM = Open Molecular Mechanics
  - Open-source library *for*
  - molecular dynamics (MD) *on*
  - HPC architectures
- Developed by Vijay Pande's group in Stanford in collaboration with Simbios (and others including CBR, SU)

Yet another MD package?

MD problem →

- AMBER
- CHARMM
- GROMACS
- HOOMD
- LAMMPS
- NAMD
- TINKER
- MyFooMD
- YourBarMM
- **OpenMM ?**

Yet another MD package?

MD problem → AMBER, CHARMM, GROMACS, HOOMD, LAMMPS, NAMD, TINKER, MyFooMD, YourBarMM → **OpenMM**

Why OpenMM?

- Dozens of MD packages
  - multiple implementations → overlapping functionality
  - new advances → needs multiple times the implementation effort
- Common ground: everybody needs speed
- What OpenMM proposes is:
  - hardware accelerated MD algorithms
  - accessed through an API
  - acting as backend for existing packages

Goals

- Ease the use of acceleration and MD
  - take advantage of various HPC architectures without porting existing code e.g. streaming architectures, clusters, etc.
  - use MD in existing code without reinventing the wheel
- Provide common interface
  - for implementing new MD features
  - for porting to new HPC platforms
- Implement multiple levels of abstraction
  - for a hardware platform independent code on the level of API
- Provide a “complete” library
  - complete = supporting the most common features

API Design Principles

- Simplicity:
  - easy to understand
  - easy to use and incorporate into existing code
- Extensibility:
  - new hardware platforms
  - new methods, algorithms
- Narrowness in scope:
  - focusing on accelerating MD algorithms only (e.g. no I/O operations)
- Hardware independence:
  - switch hardware without modifying the code (API calls)

- Language: C++

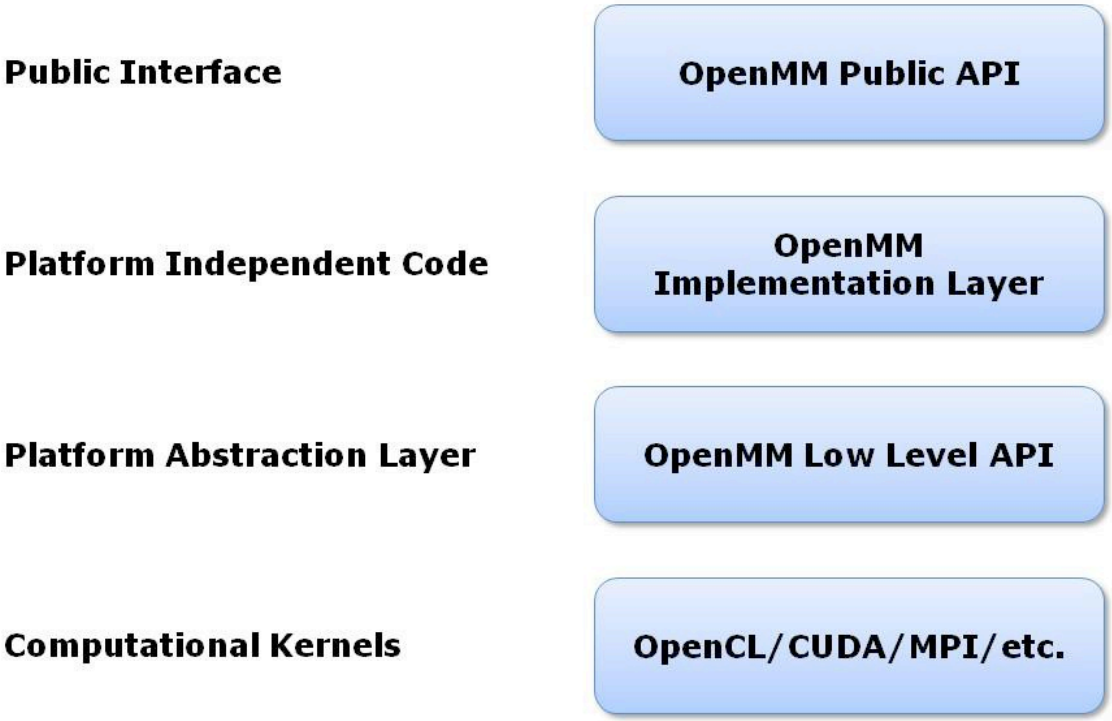
Current Capabilities

- v1.0beta
  - Platforms: Cuda, OpenCL, Brook+, Reference
  - Integrators: Langevin, Verlet, Brownian, Variable Langevin, Variable Verlet
  - Andersen thermostat
  - Center of mass removal
  - Constraints: shake, settle, ccma
  - Force fields: AMBER, Amoeba

Summary

- What **is** OpenMM
  - API and library for MD
  - Emphasis on
    - speed
    - generality
  - Open-source (mixed, mostly LGPL)
- What **isn't** OpenMM
  - Full-fledged MD package - no I/O and analysis
  - General solution for **all** MD problems
  - Magic
    - not a cross compiler
    - no protection from specific hardware quirks (e.g. frequent access to simulation data on GPUs will cause slow execution – PCI-e bottleneck)

OpenMM Architecture



Platform Dependent Code

- Computational kernels
  - currently consists of CUDA, Brook, and OpenCL kernels

- Low Level API
  - abstraction layer to hide the platform details
  - stream computing formulation:
    - **Stream**: opaque reference to an array of data
    - **Kernel**: calculation on streams
  - **Platform**:
    - defines a set of **Kernel**s and **Stream**s
    - each platform packaged as a dynamic library
    - automatic platform selection (fastest available)

Platform Independent Code

- Implementation Layer
  - platform independent code, interface between the Public API and lower level, platform specific code
- Public API
  - classes that expose the functionalities of the library
    - **System**
      - a collection of interacting particles with mass, constraints, etc.
    - *"***Force***"*
      - anything that affects the system behavior, e.g. forces, thermostats, etc.
    - **Context**
      - system state informations, e.g. positions, velocities, other data etc.
    - **Integrator**
      - advances the system in time (currently only fixed step size)
    - **State**
      - snapshot of the system's state at certain time; the only way to access positions, velocities, etc.

OpenMM in Use

(HelloArgon OpenMM example)

```
// Create a System with nonbonded forces
OpenMM::System system;
OpenMM::NonbondedForce* nonbond = new OpenMM::NonbondedForce();
system.addForce(nonbond);

// Create three atoms.
std::vector<OpenMM::Vec3> initPosInNm(3);
for (int a = 0; a < 3; ++a)
{
    // Space atoms out evenly (in nm)
    initPosInNm[a] = OpenMM::Vec3(0.5*a,0,0);
    system.addParticle(39.95); // mass(Ar), g/mole
    // charge, L-J sigma (nm), epsilon (well depth)s (kJ)
    nonbond->addParticle(0.0, 0.3350, 0.996); // vdWRad(Ar)=.188 nm
}
```

OpenMM in Use (continued)

```
    // Set step size in ps
    OpenMM::VerletIntegrator integrator(0.004);

    // Let OpenMM Context choose best platform
    OpenMM::OpenMMContext context(system, integrator);
    // Set starting positions of the atoms; leave time and velocity zero
    context.setPositions(initPosInNm);

    // Simulate
    for (int frameNum=1; ;++frameNum) {
    // Output current state information
    OpenMM::State state = context.getState(OpenMM::State::Positions);
    const double timeInPs = state.getTime();
    writePdbFrame(frameNum, state); // output coordinates
    if (timeInPs >= 10.) // done?
        break;
    // Advance state many steps at a time, for efficient use of OpenMM
    integrator.step(10); // (use a lot more than this normally)
}
```

Gromacs-OpenMM

- Goal
  - provide GPU acceleration capabilities to Gromacs
  - `mdrun -device "openmm:platform=cuda,device=1,memtest=1"`
- Benefit – the best of the two worlds
  - Gromacs: versatile I/O and analysis tools
  - OpenMM: support for multiple GPU accelerator platforms

Gromacs-OpenMM (continued)

- Limitations
  - only a subset of Gromacs features available in OpenMM
  - fetching the system state (coordinates, velocities, forces, etc.) is a limiting factor, e.g. toy system consisting of one ethane molecule, simulated for 1ns

State retrieval frequency (fs)	2	5	10	100
Execution time (s)	100	60	49	35

Integration quirks

- Feature mapping
  - which Gromacs features/options correspond to which OpenMM features
  - careful mapping to avoid unexpected results

- Hardware reliability
  - experience shows that GPUs often have faulty memory
    - especially consumer-level devices
    - heat, overclocking, insufficient or unstable power supply
  - Gromacs-OpenMM does a pre- and post-simulation memory testing
    - uses memtestG80
    - not a bulletproof solution but catches seriously misbehaving hardware

Availability

- Gromacs-OpenMM release with Gromacs 4.1
  - **experimental**
  - CUDA support
  - December 2009

- Links
  - https://simtk.org/home/openmm
  - http://www.gromacs.org/