RUMD.org:
A molecular dynamics code optimized for GPUs

Roskilde University Molecular Dynamics

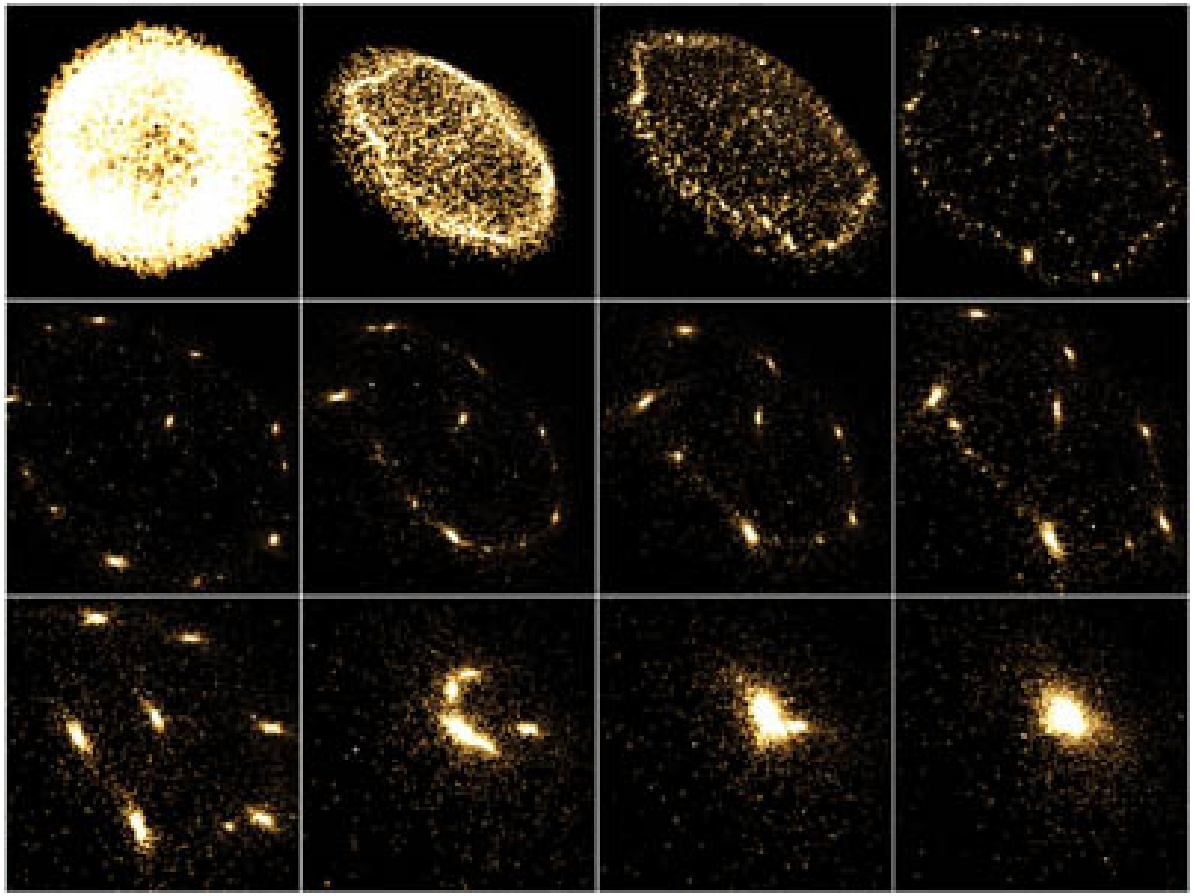
Thomas Schrøder,
Glass & Time, NSM, RUC

GPU Computing Today and Tomorrow
GPU-Lab, DTU, 18/8-2011

Glass and Time

Danish National Research Foundation Centre for Viscous Liquid Dynamics

The Nbody program:
Simulating galaxies interacting by gravitational forces



$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\left\|\mathbf{r}_{ij}\right\|^2}\cdot\frac{\mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|},$$

$$\mathbf{F}_i = \sum_{\substack{1\le j\le N \\ j\neq i}} \mathbf{f}_{ij} = Gm_i \cdot \sum_{\substack{1\le j\le N \\ j\neq i}} \frac{m_j \mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|^3}.$$

1) Calculate forces and accelerations, O(N^2)
2) Advance positions and velocities one timestep using Newton's equatios O(N).
3) Goto 1

Molecular Dynamics (MD)

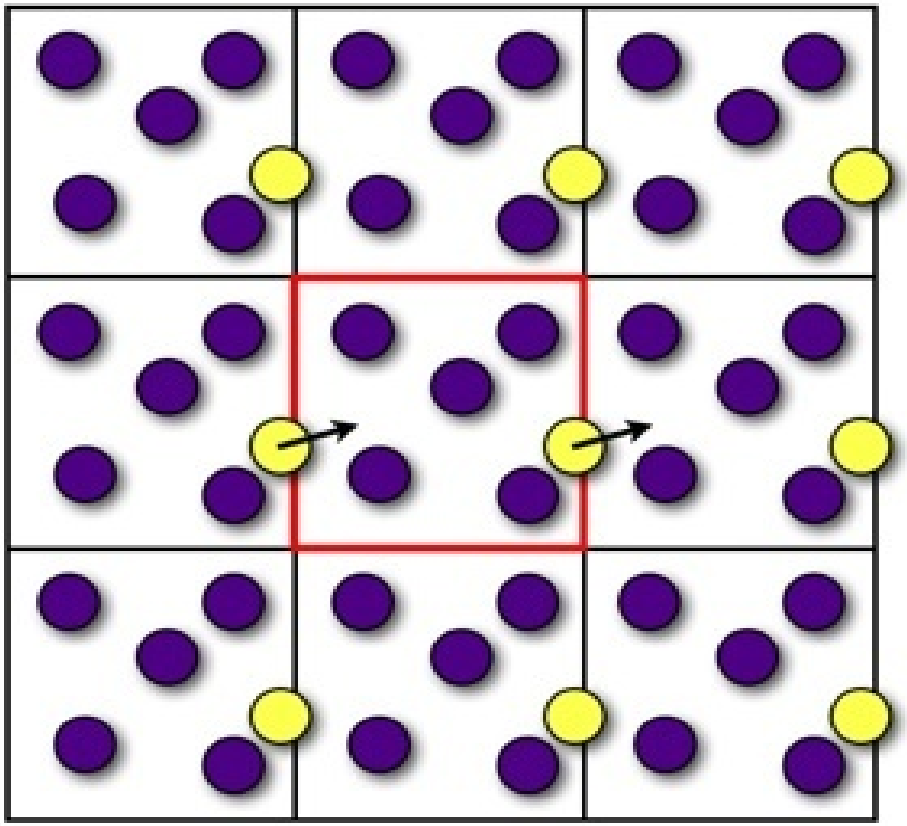
Lennard-Jones pair potential:

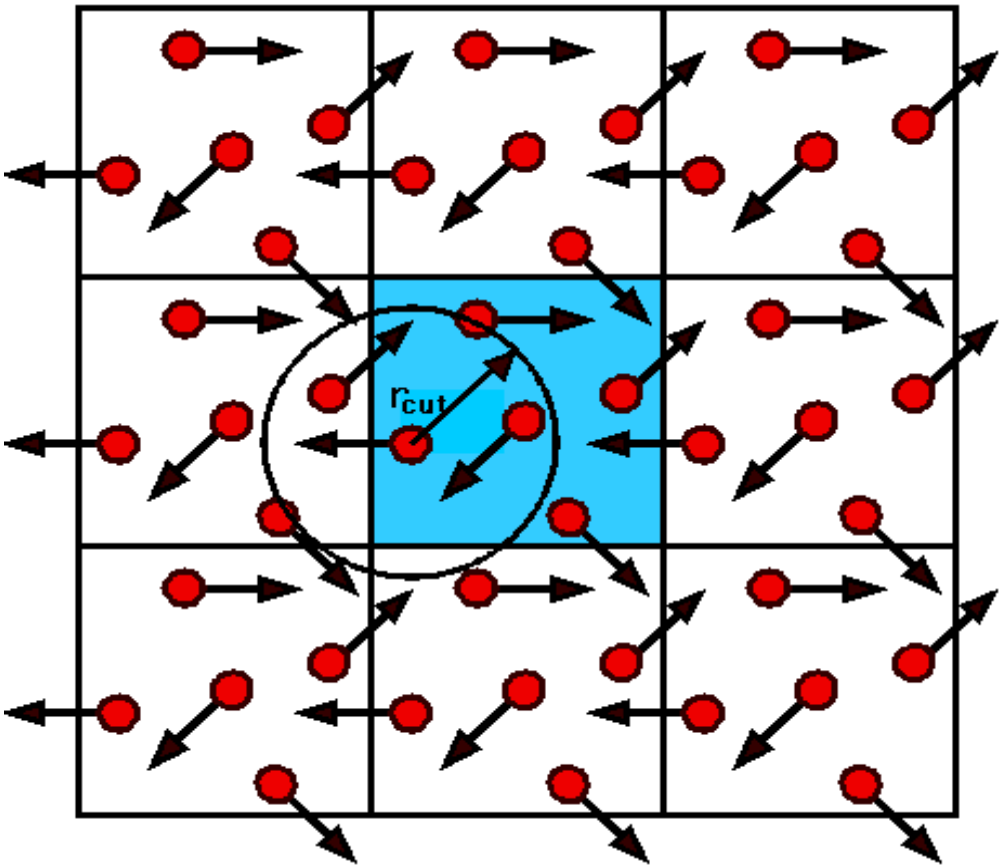
$$\Phi_{\alpha\beta}(r) = 4\epsilon_{\alpha\beta}\left(\left(\frac{\sigma_{\alpha\beta}}{r}\right)^{12} - \left(\frac{\sigma_{\alpha\beta}}{r}\right)^{6}\right)$$

Like Nbody, except:

Periodic boundary conditions

Cut-off in potential (nb-lists) and minimal image convention





Simulations are 3D

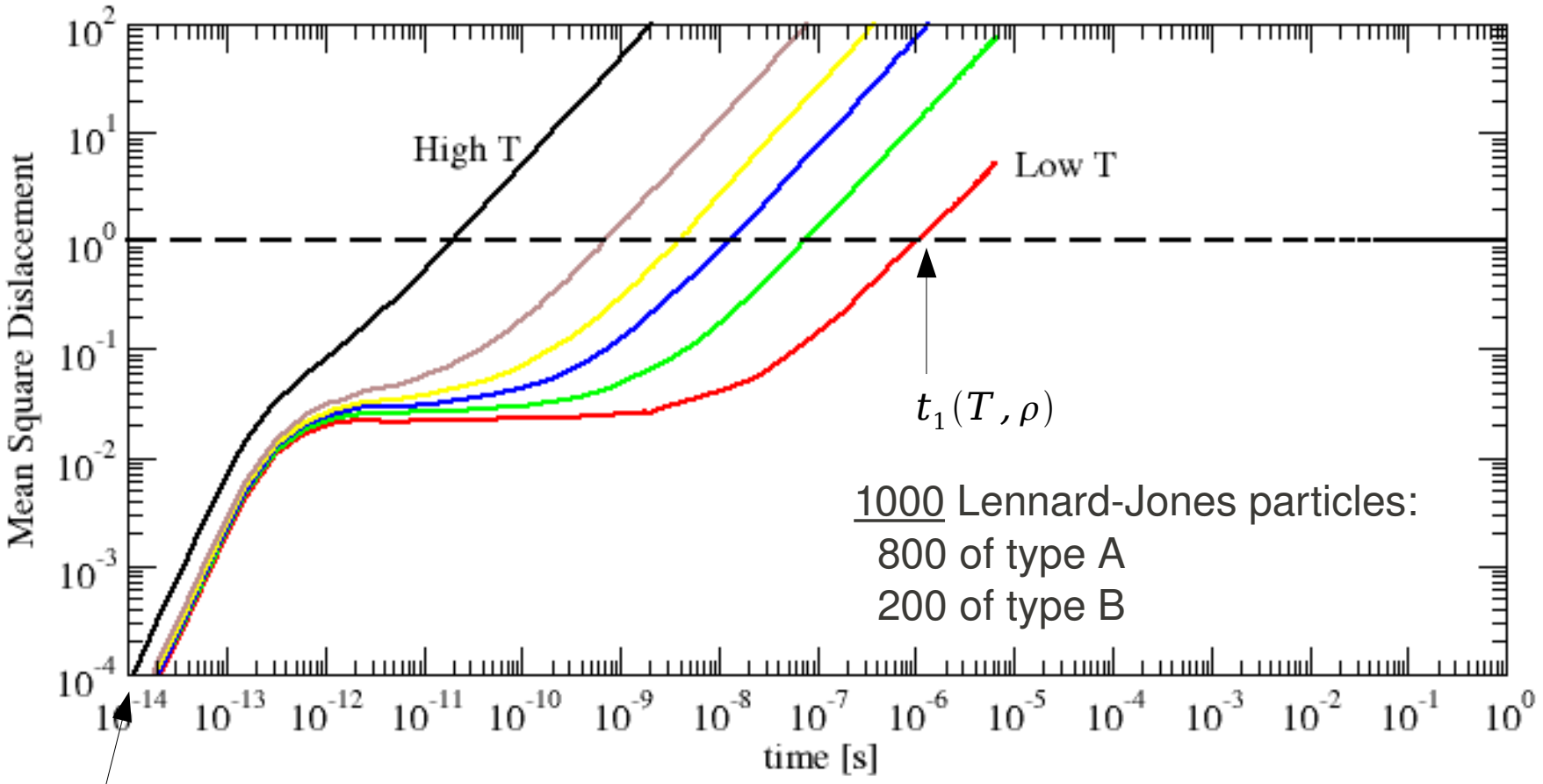
Particles inside cut-off: ~ 100

Nbody: O(N^2), regular, static

MD: O(N), irregular, dynamic

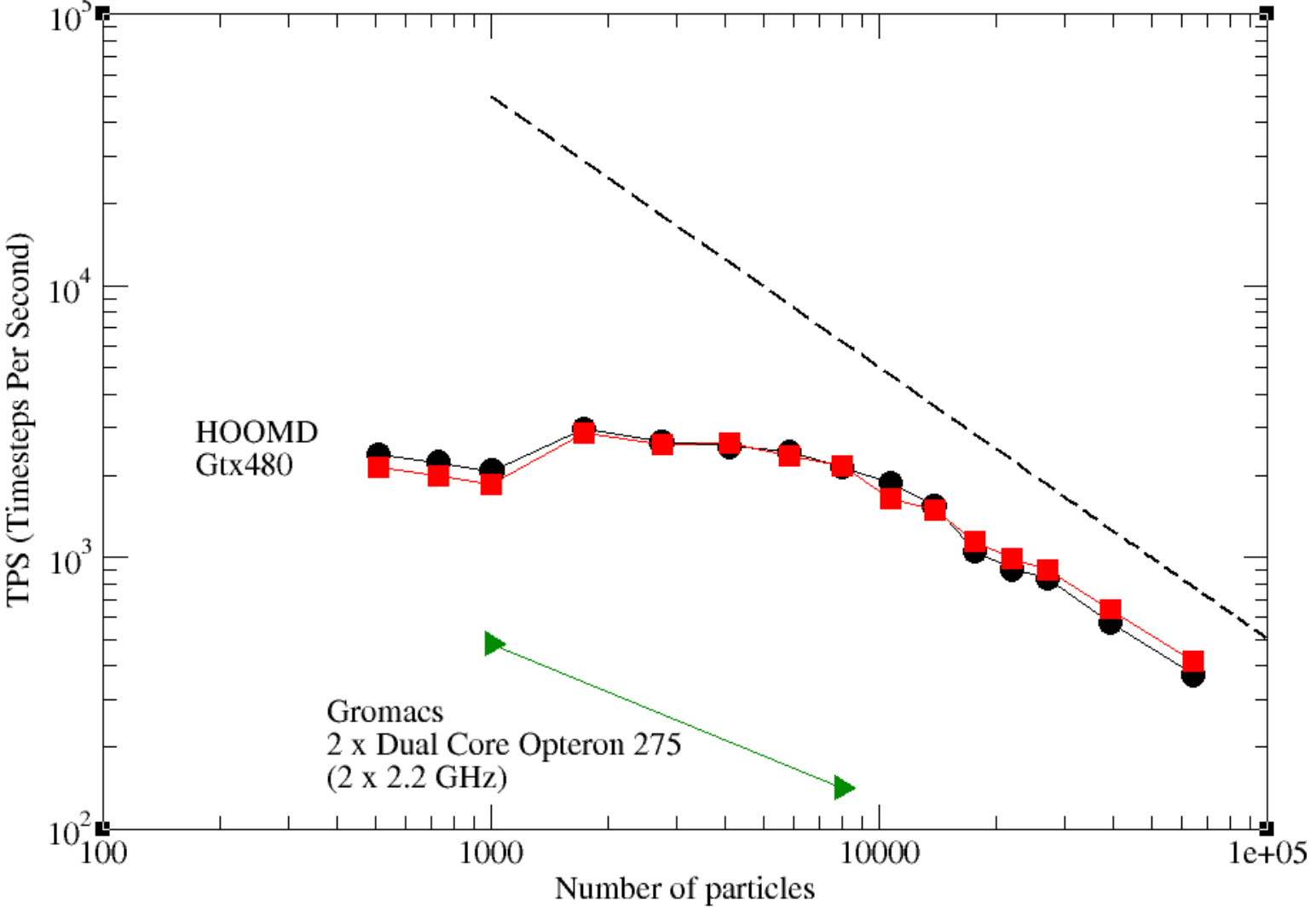
The need for speed:

To test theories and investigate new phenomena, we want to simulate liquids on the millisecond timescale and beyond.



Size of our time-step

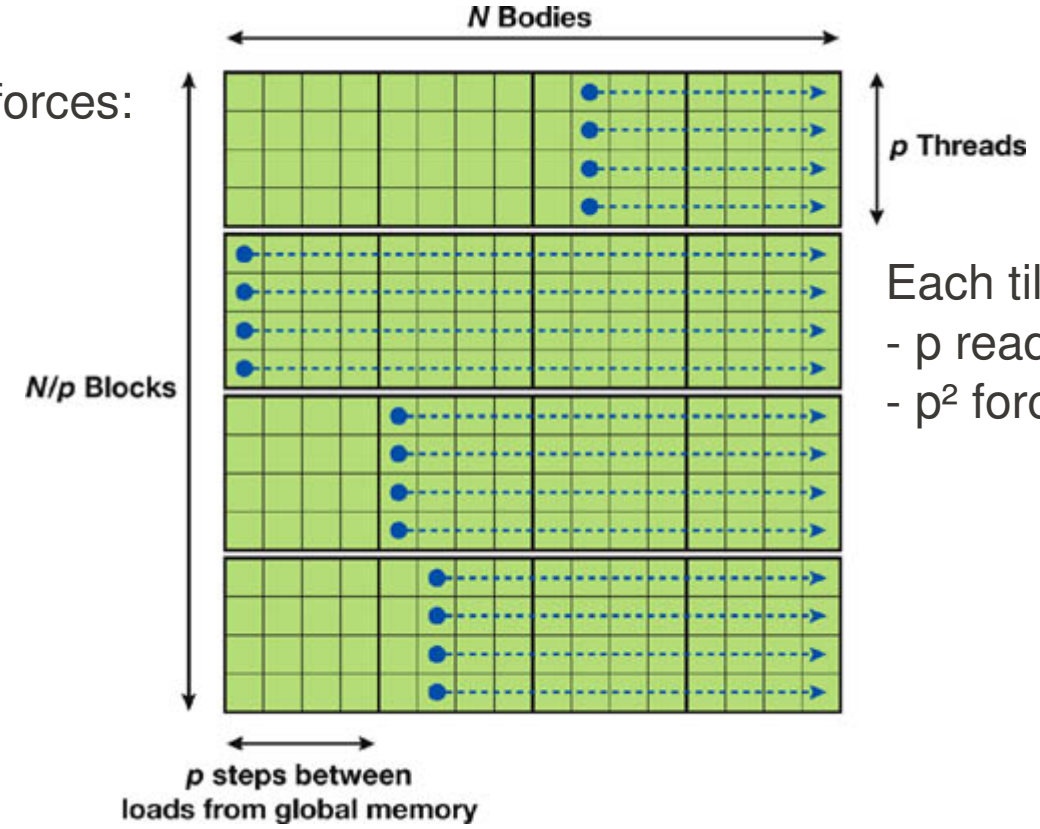
CPU versus GPU
(4 versus 480 cores)



Challenge: Getting good speed-up when samples are small (N~1000)

The overall structure of force calculations in Nbody (and RUMD):

The N*N forces:

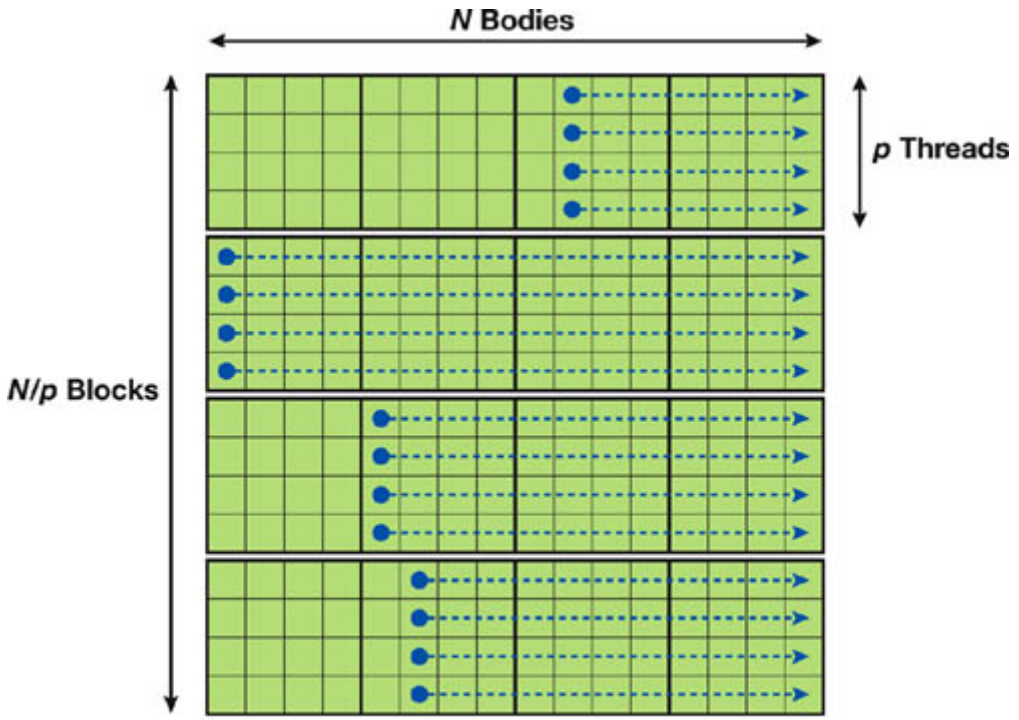


Each tile:
- p reads from device memory
- p^2 force calculations

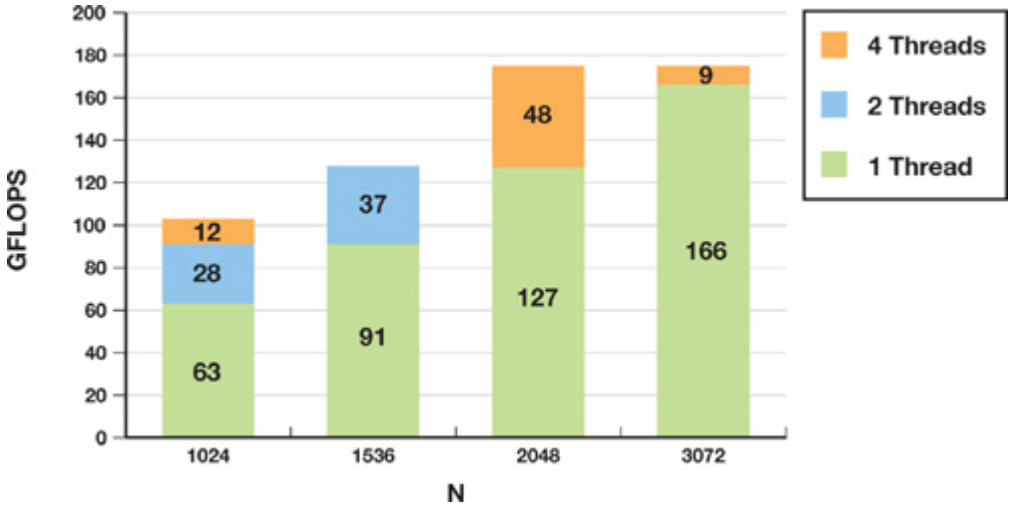
Optimizing for small samples:

Using several threads per particle

-> better latency hidding



Thread: 0 1 0 1 (2 threads per part.)



Strategy for RUMD implementation:

Optimize for N=1000

Worry about good scaling later

First step:

Re-implementing Nbody [O(N^2)].

4 times faster than highly optimized CPU MD program [O(N)], for N=1000 !!!!

Second step:

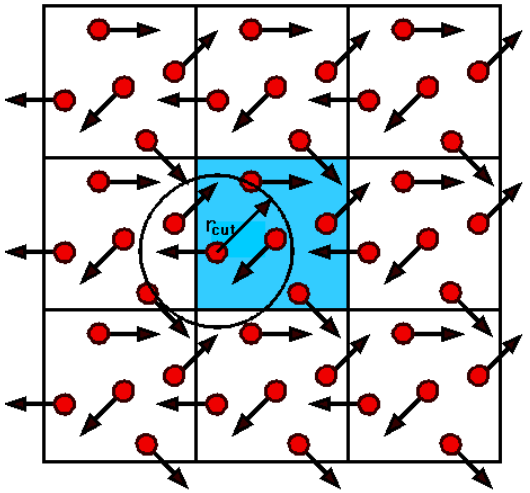
Changing interactions to multi-component Lennard-Jones

(+ other molecular dynamics features)

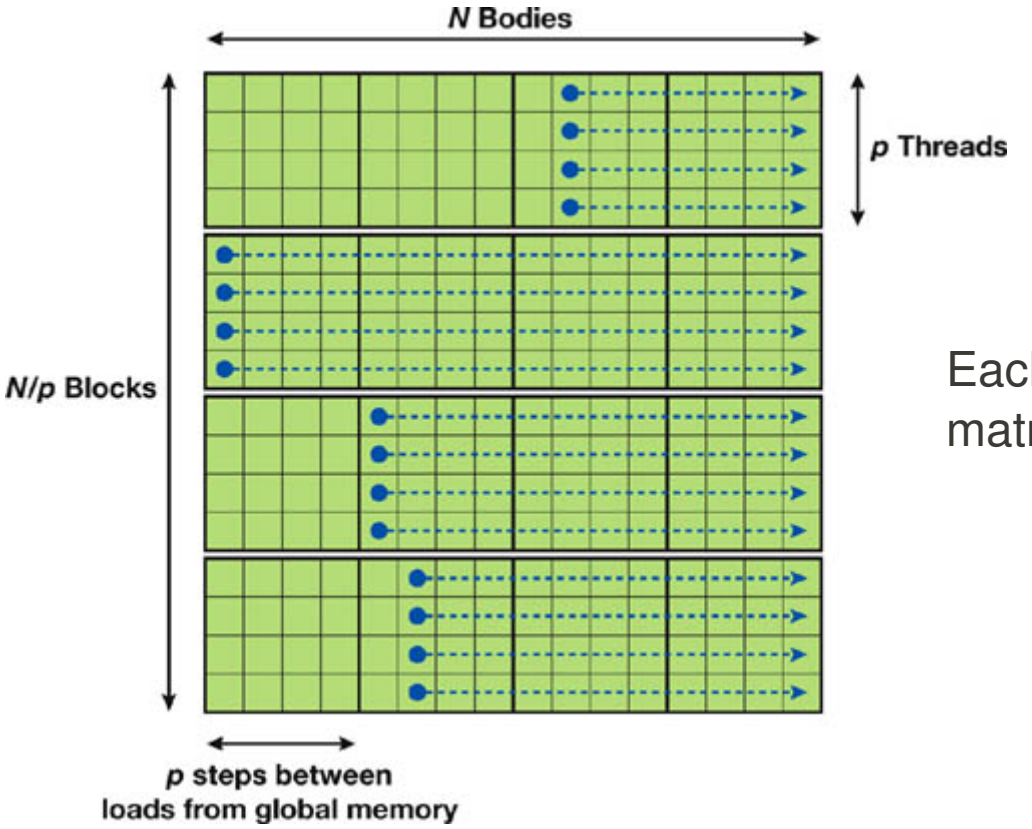
2 times faster than highly optimized CPU MD program [O(N)], for N=1000

Third step: Adding 'Neighbour-lists'

- making force calculation O(N) instead of O(N^2)



Actually neighbour-matrix:



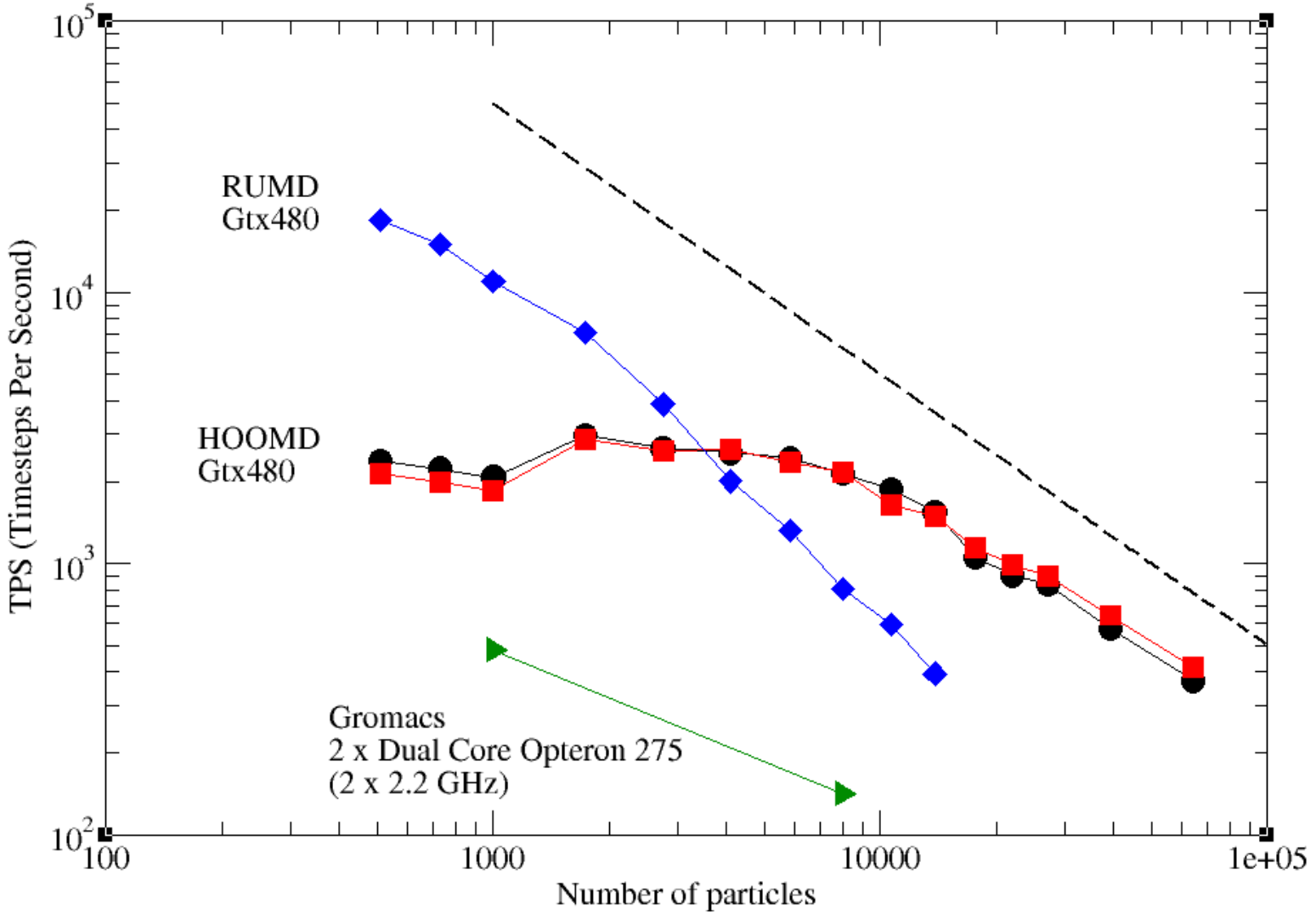
Each tile has a (sub-) Neighbour-matrix, encoded as (32 bit) integers:

0010 ...
0101
1000
1001
.
.
.

Neighbour matrix decoded by:

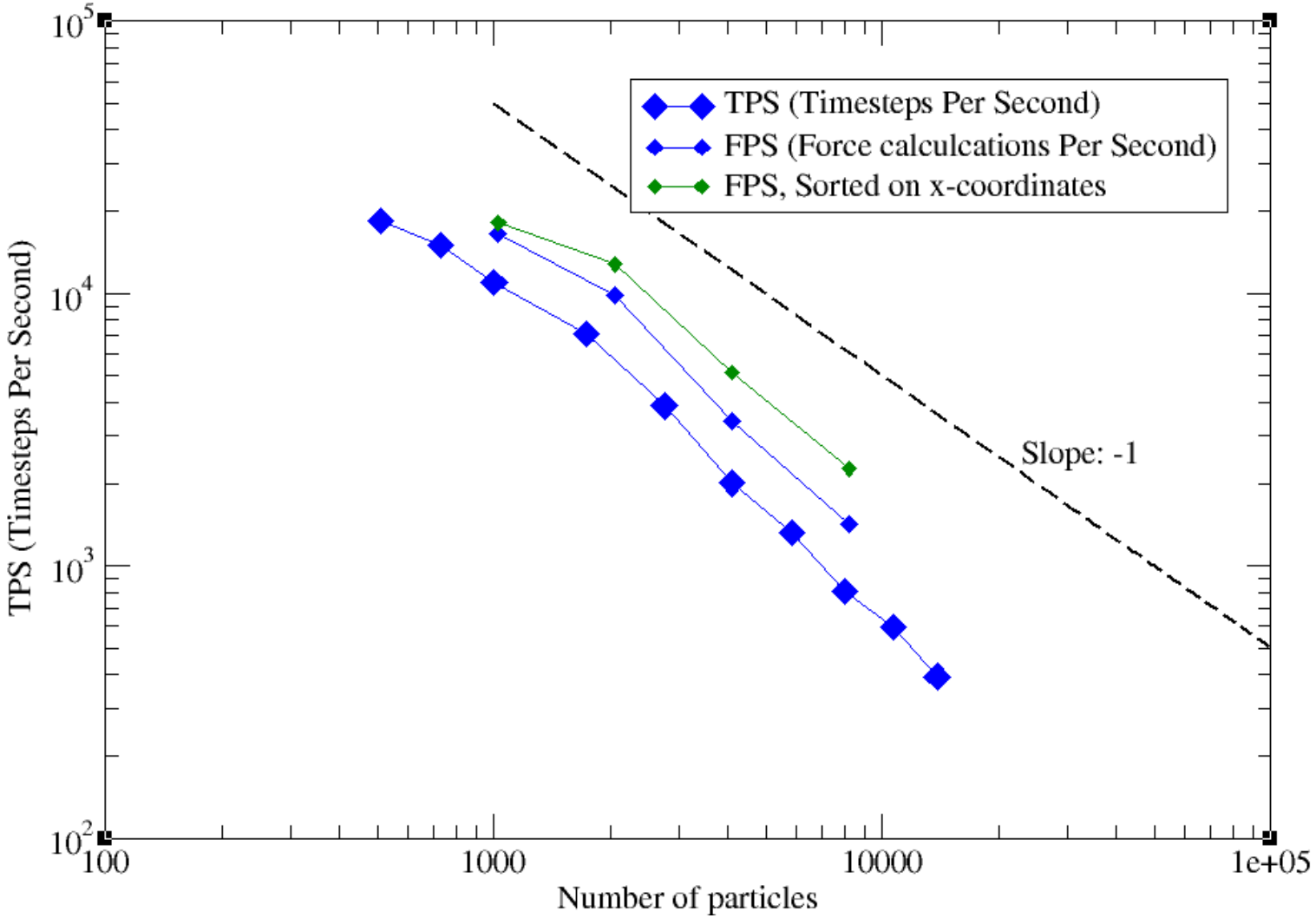
`__ffs(x)` returns the position of the first (least significant) bit set in integer parameter **`x`**. The least significant bit is position 1. If **`x`** is 0, **`__ffs()`** returns 0.

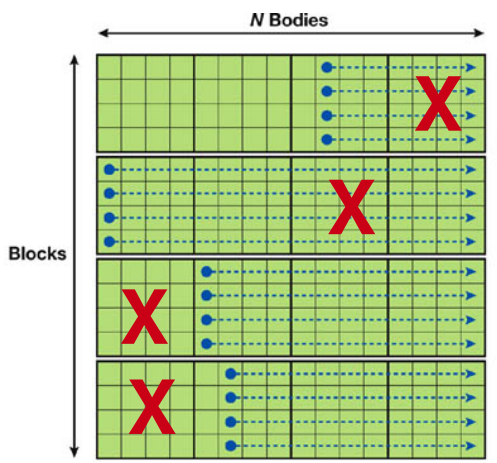
Benchmarks



For N=1000 RUMD 23 times faster than Gromacs (4 optoron cores)
5 times faster than HOOMD (same Gtx480)

Improving scaling by sorting in 3D space (experimental)





Each block has
a list of which other
blocks it interacts with

Plus a number of other features (see http://rumd.org):

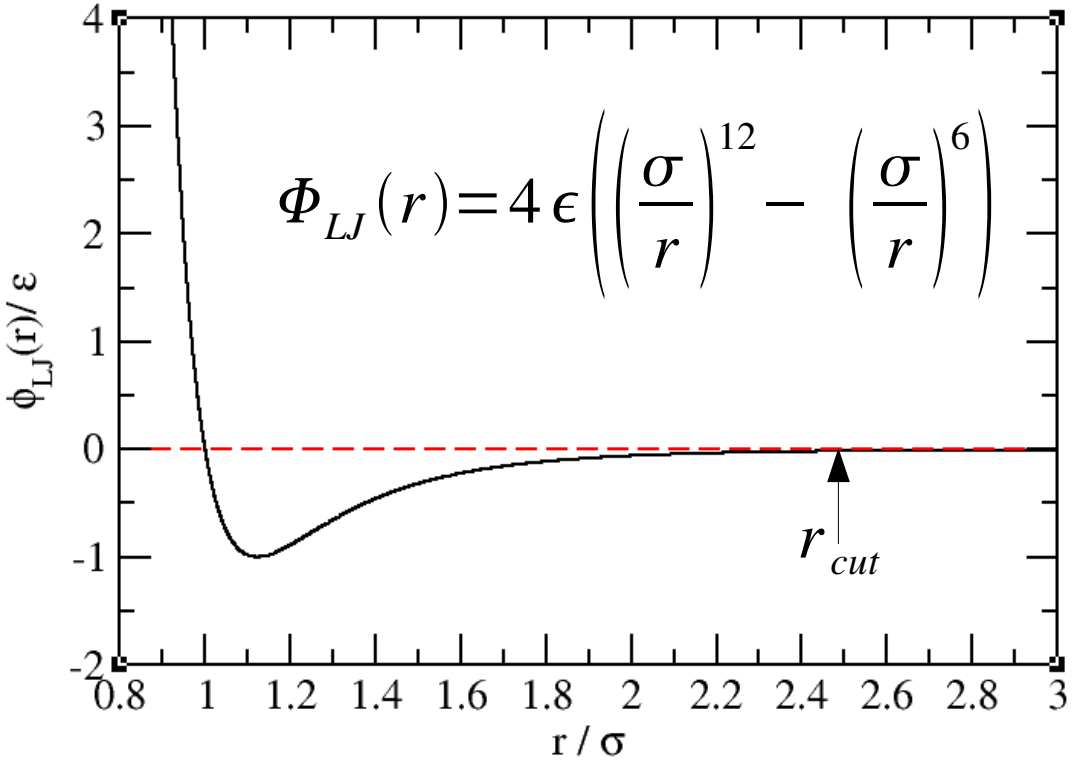
- van der Waals type pair potentials: Lennard-Jones, Gaussian core, Inverse Power Law, and more. It is easy to implement new pair potentials.
- Bond stretching potentials: Harmonic and FENE
- Multicomponent simulations
- NVE and NVT ensemble simulations
- python interface ◄
- Post simulation analysis tools

```python
import sys
sys.path.insert(0,"../Swig")

from rumdSimulation import rumdSimulation, rumd

# Setup simulation with initial configuration
sim = rumdSimulation("start.xyz.gz")

# Choose integrator
itg = rumd.IntegratorNVT(timeStep=0.002, targetTemperature=0.5)
#itg = rumd.IntegratorNVE(timeStep=0.0025)
sim.SetIntegrator(itg)

# Create Lennard-Jones pair-potential
potential = rumd.Pot_LJ_12_6()

# Set up parameters for Lennard-Jones interactions ( Kob & Andersen )
# i  j   Sig   Eps  Rcut (in units of Sigma_ij)
potential.SetParams(0, 0, 1.00, 1.00, 2.5)
potential.SetParams(0, 1, 0.80, 1.50, 2.5)
potential.SetParams(1, 0, 0.80, 1.50, 2.5)
potential.SetParams(1, 1, 0.88, 0.50, 2.5)
sim.SetPotential(potential)

sim.Run(30000000)

# Save the final configuration
sim.sample.WriteConf("end.xyz.gz")
```


The RUMD developers:
Nicholas Bailey,
Trond Ingebrigtsen,
Jesper S. Hansen,
Lasse Bøhling,
Heine Larsen,
Thomas Schrøder

Thank you for your attention

The Lennard-Jones potential:

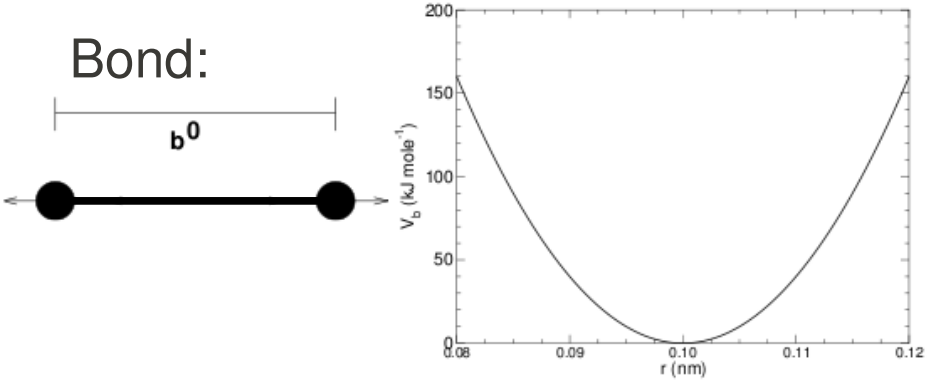


Other non-bonded pair-interactions:

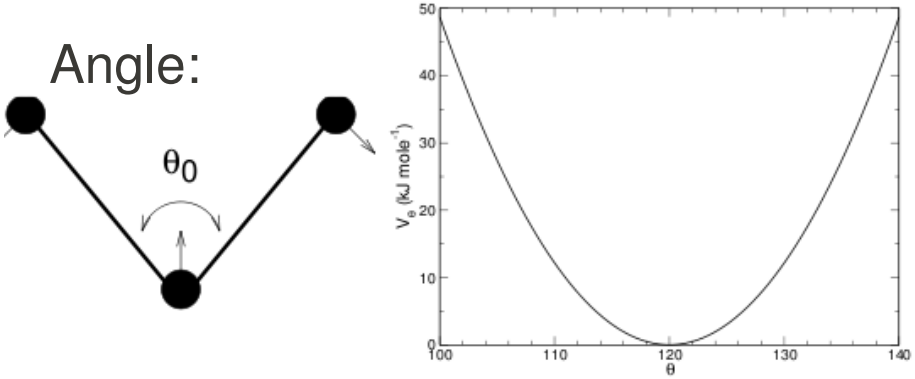
- Coulomb
- Soft spheres: $\phi(r) \propto r^{-n}$
- ...
- ...
- ...

Bonded interactions:

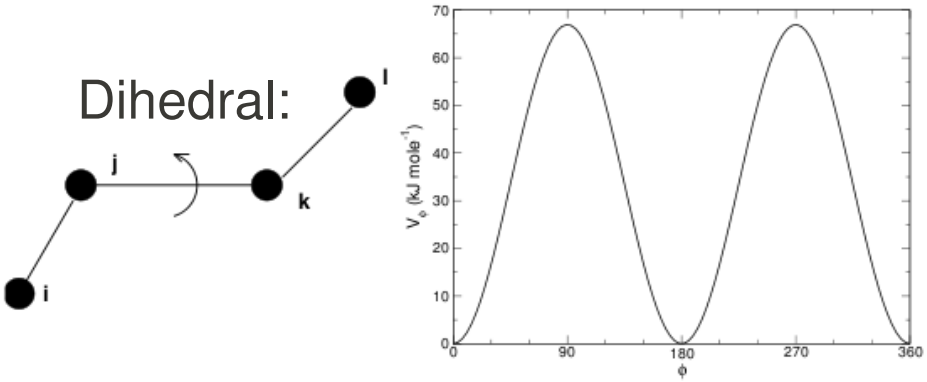
Bond:

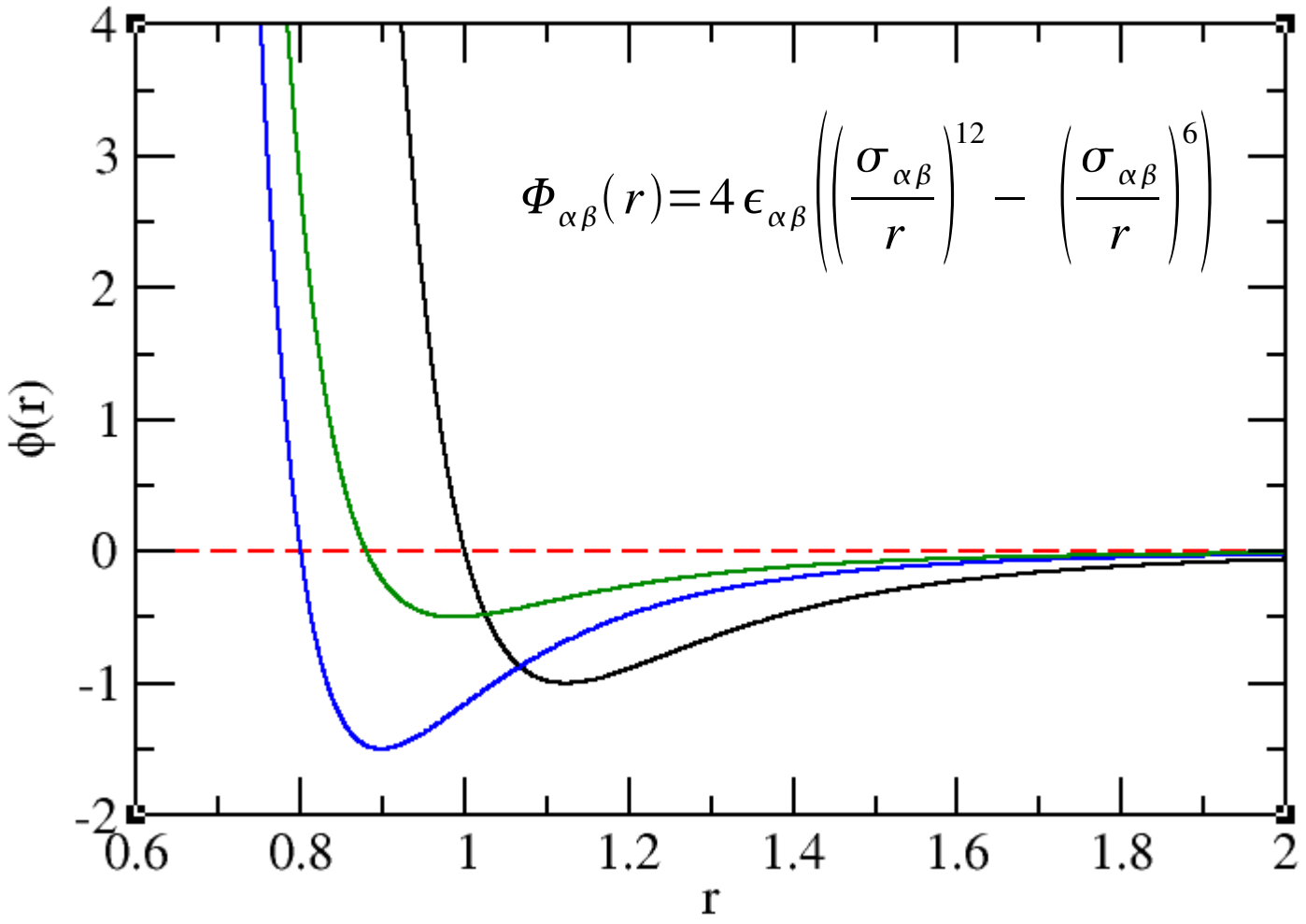


Angle:



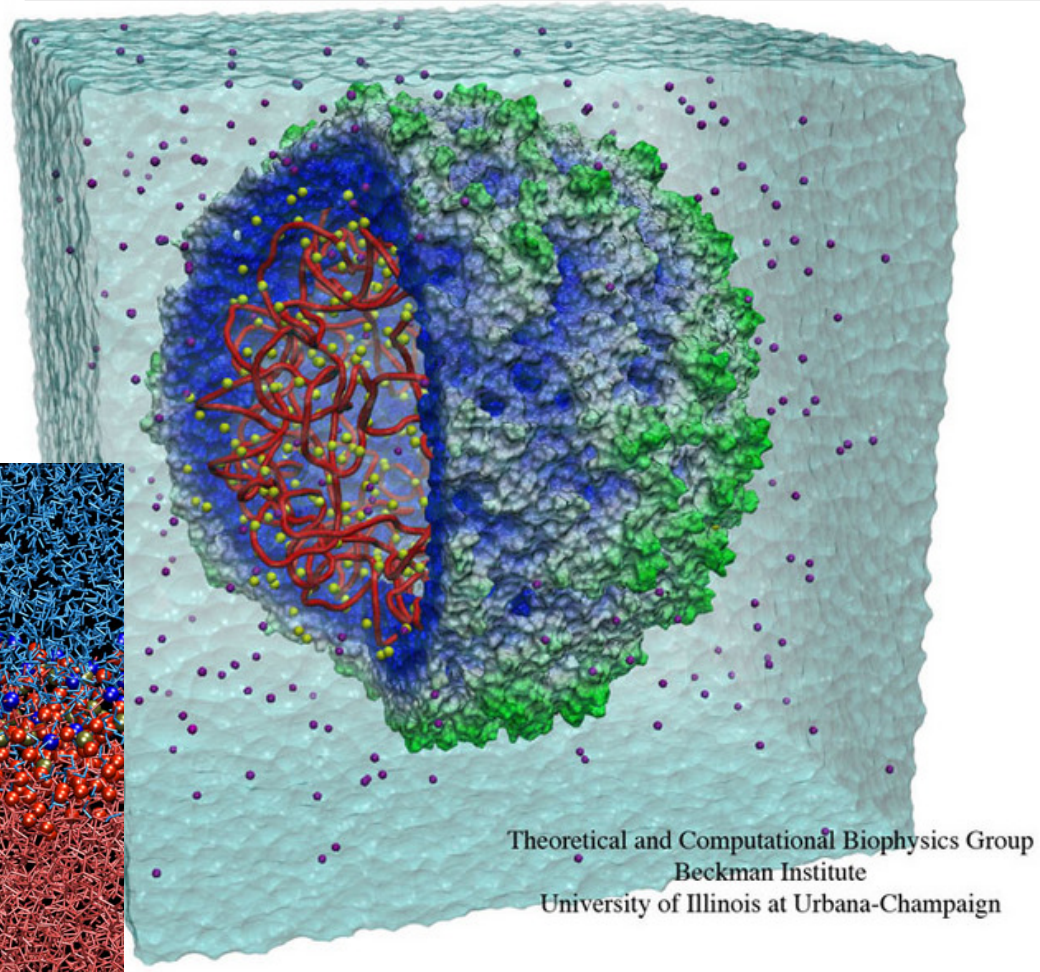
Dihedral:



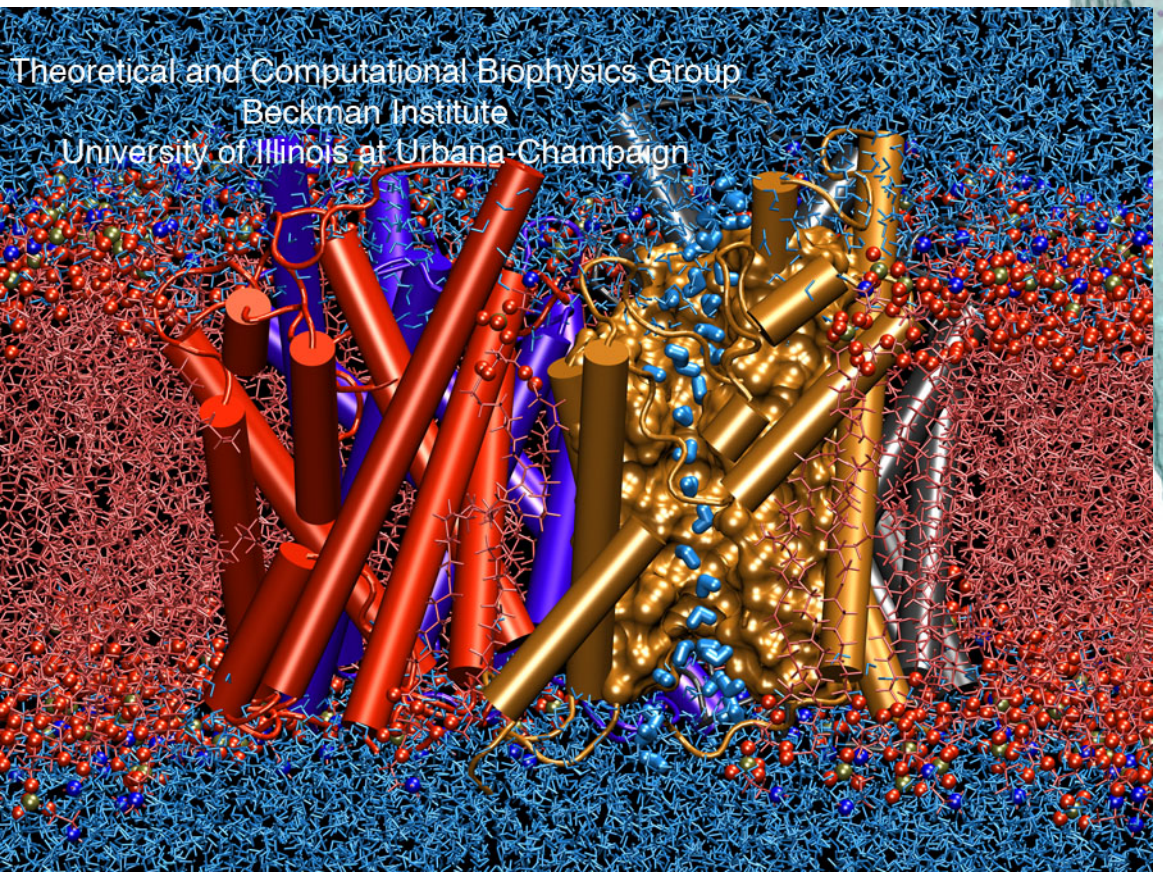

$\Phi_{\alpha\beta}(r) = 4\epsilon_{\alpha\beta}\left(\left(\frac{\sigma_{\alpha\beta}}{r}\right)^{12} - \left(\frac{\sigma_{\alpha\beta}}{r}\right)^{6}\right)$
ϕ(r)
r
4
3
2
1
0
-1
-2
0.6
0.8
1
1.2
1.4
1.6
1.8
2

From these building blocks very complex molecules can be simulated:

satellite tobacco mosaic virus, complete with protein, RNA, ions, and a small water box



Water permeation through membrane water channels

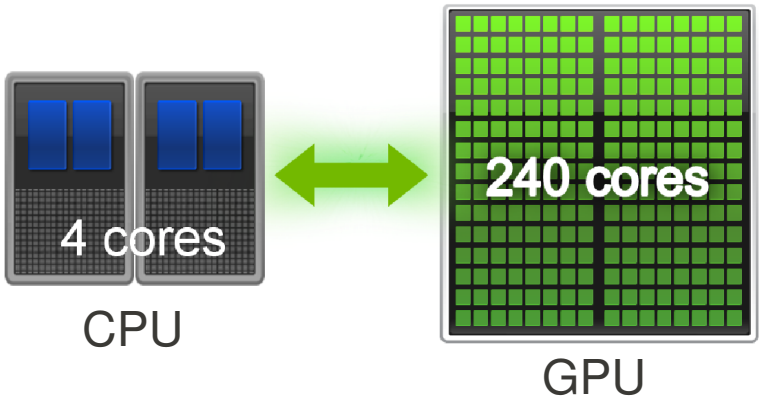


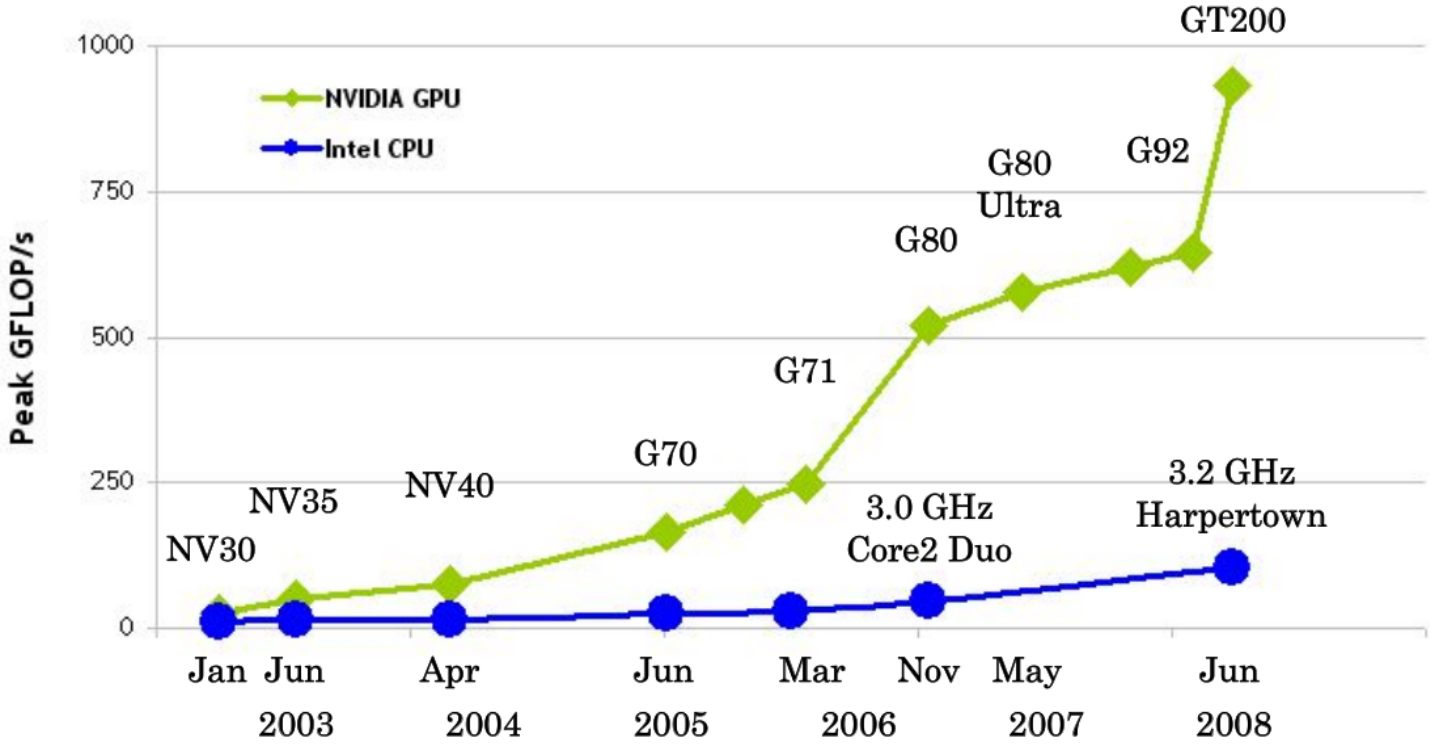
[http://www.ks.uiuc.edu/Gallery/]

The Glass and Time GPU Cluster

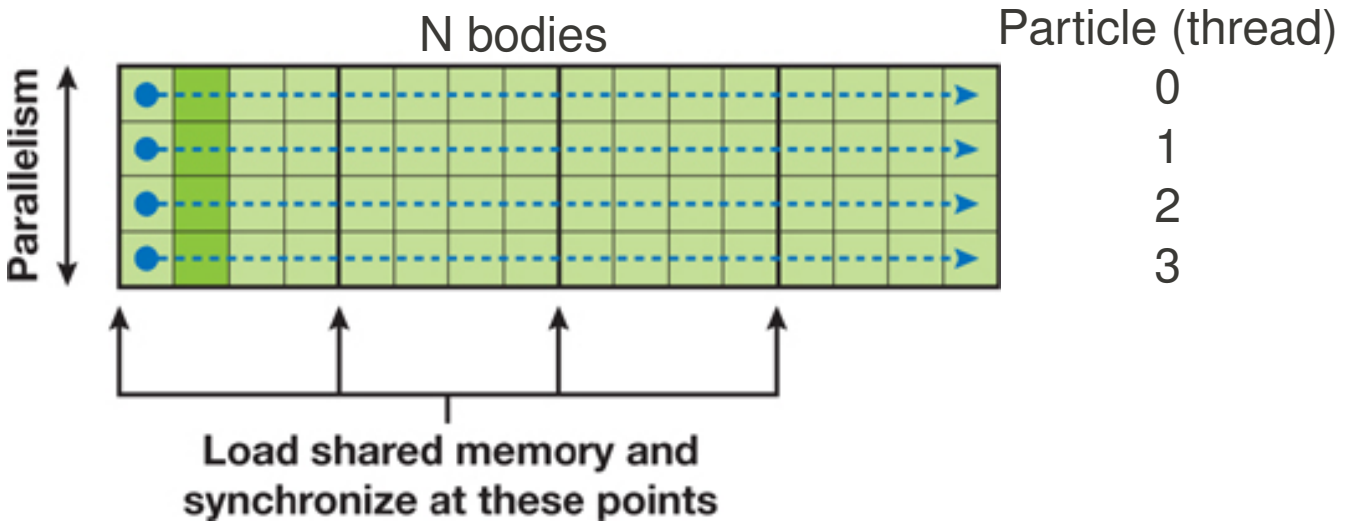
92 GPU's (mostly Gtx280)
Total theoretical peak perfomance: ~ 80 TFlops

CPU vs. GPU





The new Glass & Time cluster: 92 GPU's (GT200). Total peak perfomance: 85.8 TFlops.




```
01.     __global__ void
02. calculate_forces(void *devX, void *devA)
03. {
04.   extern __shared__ float4[] shPosition;
05.   float4 *globalX = (float4 *)devX;
06.   float4 *globalA = (float4 *)devA;
07.   float4 myPosition;
08.   int i, tile;
09.   float3 acc = {0.0f, 0.0f, 0.0f};
10.   int gtid = blockIdx.x * blockDim.x + threadIdx.x;
11.   myPosition = globalX[gtid];
12.   for (i = 0, tile = 0; i < N; i += p, tile++) {
13.     int idx = tile * blockDim.x + threadIdx.x;
14.     shPosition[threadIdx.x] = globalX[idx];
15.     __syncthreads();
16.     acc = tile_calculation(myPosition, acc);
17.     __syncthreads();
18.   }
19.   // Save the result in global memory for the integration step.
20.    float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
21.   globalA[gtid] = acc4;
22. }
```

Loop over tiles

Read to shared memory

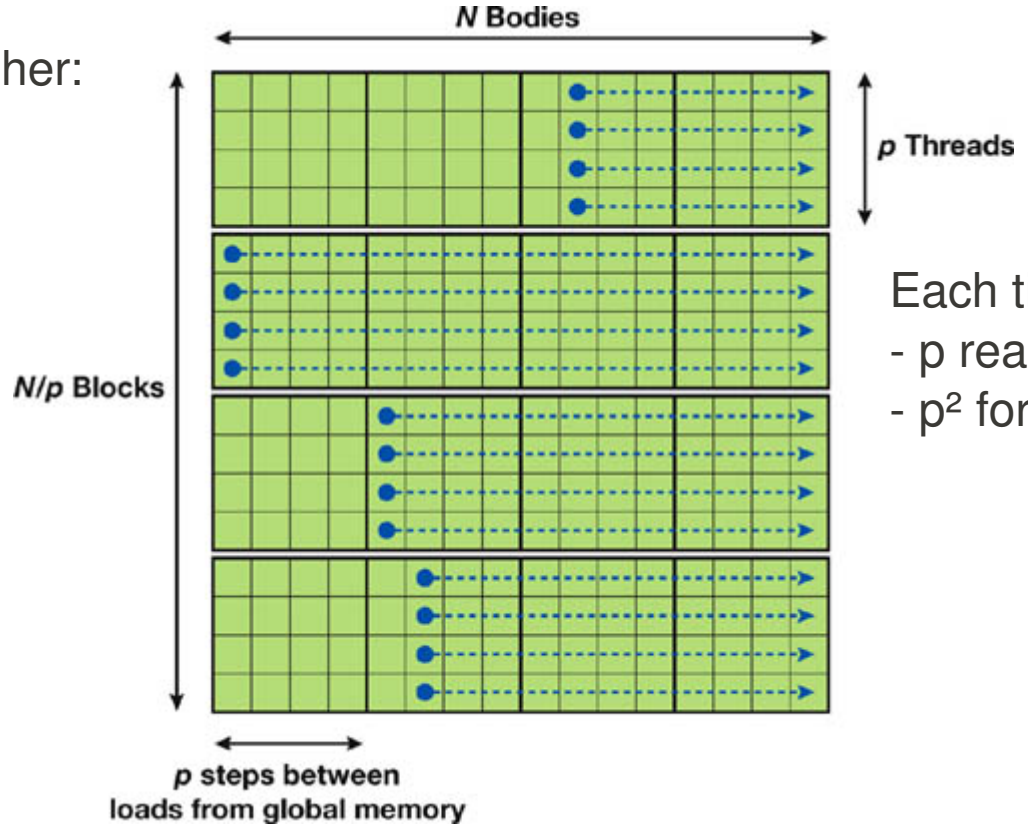
Do 'my' part of tile

```
    __device__ float3
tile_calculation(float4 myPosition, float3 accel)
{
  int i;
  extern __shared__ float4[] shPosition;
  for (i = 0; i < blockDim.x; i++) {
    accel = bodyBodyInteraction(myPosition, shPosition[i], accel);
  }
  return accel;
}
```

```
    __device__ float3
bodyBodyInteraction(float4 bi, float4 bj, float3 ai)
{
  float3 r;
  // r_ij  [3 FLOPS]
  r.x = bj.x - bi.x;
  r.y = bj.y - bi.y;
  r.z = bj.z - bi.z;
  // distSqr = dot(r_ij, r_ij) + EPS^2  [6 FLOPS]
   float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;
  // invDistCube =1/distSqr^(3/2)  [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
   float distSixth = distSqr * distSqr * distSqr;
  float invDistCube = 1.0f/sqrtf(distSixth);
  // s = m_j * invDistCube [1 FLOP]
   float s = bj.w * invDistCube;
  // a_i =  a_i + s * r_ij [6 FLOPS]
  ai.x += r.x * s;
  ai.y += r.y * s;
  ai.z += r.z * s;
  return ai;
}
```

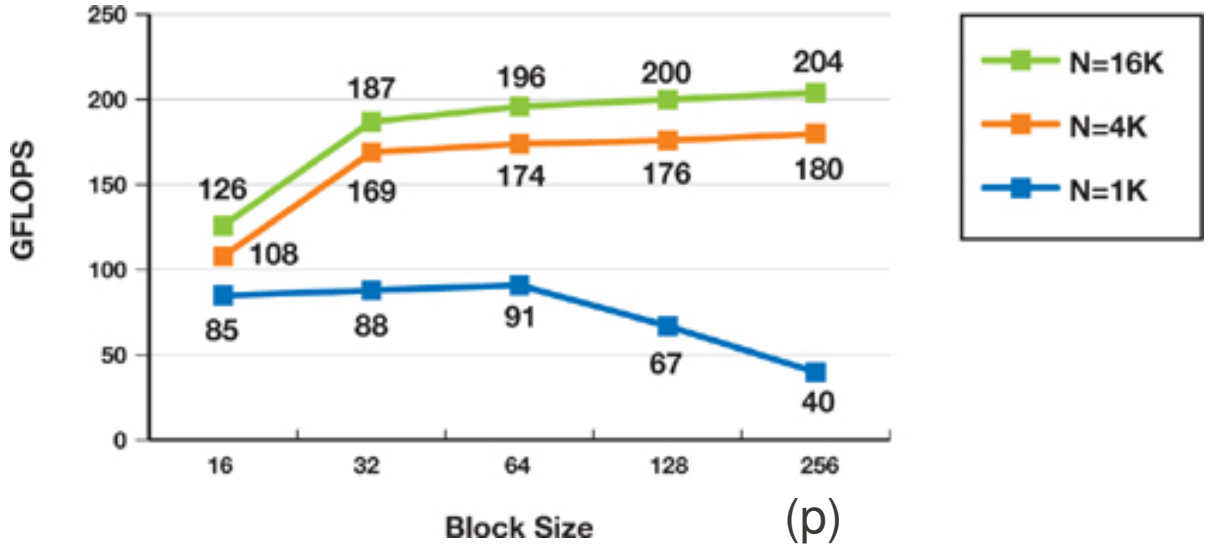
$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2}\cdot\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|},$$

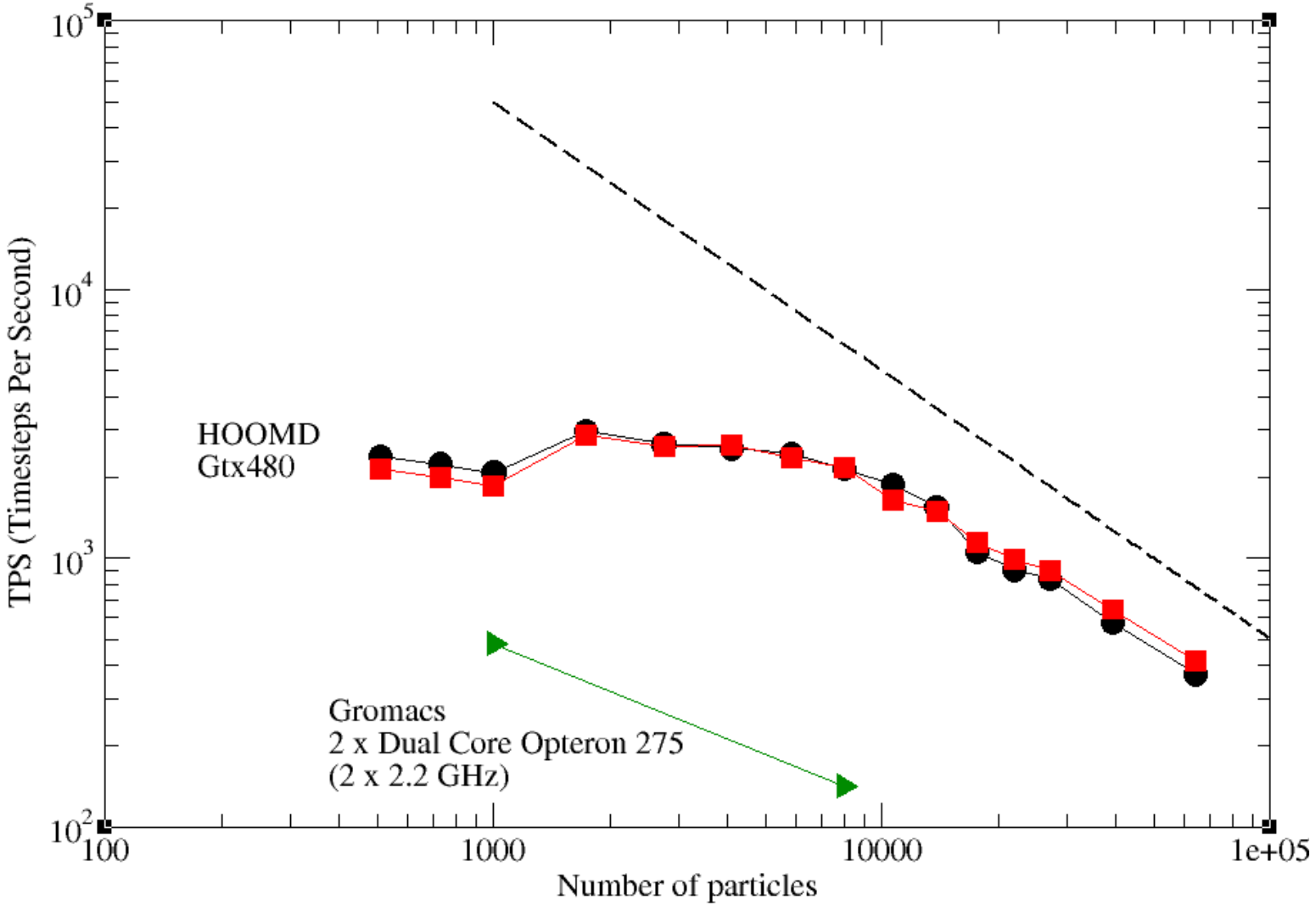
Putting it all together:



Each tile:
- p reads from device memory
- p^2 force calculations

Full speed (GTX 8800):
~ 200GFlops




10^5
10^4
10^3
10^2
TPS (Timesteps Per Second)
HOOMD
Gtx480
Gromacs
2 x Dual Core Opteron 275
(2 x 2.2 GHz)
100
1000
10000
1e+05
Number of particles

Corresponding kernel in RUMD:

```
template<int STR, int CONFT, class P, class S>
__device__ __host__ float fij( P* Pot, float4 my_r, float4 rj, float4* my_f, float4* my_w,
float4* my_sts, float4* my_misc, float* param, S* simBox, float* simBoxPointer ){

  float4 dist = simBox->calculateDistance(my_r, rj, simBoxPointer);

  // Inside cut-off for interaction? (param[0]=Rcut^2)
  if ( dist.w <= param[0] && dist.w >= 0.000001f ){
    float s;

    if(CONFT){
      s = Pot->ComputeInteraction(dist.w, param, my_f, my_w, my_misc);
    }
    else{
      s = Pot->ComputeInteraction(dist.w, param, my_f, my_w);
    }

    (*my_f).x += dist.x * s;
    (*my_f).y += dist.y * s;
    (*my_f).z += dist.z * s;

    if(STR){
      // stress - diagonal components
      (*my_sts).x -= dist.x * dist.x * s;  // xx
      (*my_sts).y -= dist.y * dist.y * s;  // yy
      (*my_sts).z -= dist.z * dist.z * s;  // zz
      // stress - off-diagonal components
      (*my_sts).w -=  dist.y * dist.z * s; // yz
      (*my_w).y -=  dist.x * dist.z * s;   // xz
      (*my_w).z -=  dist.x * dist.y * s;   // xy
    }
  } // END  dist.w <= param[0]...

  return dist.w;
}
```

Periodic Boundary Conditions (prepared for other shaped boxes)

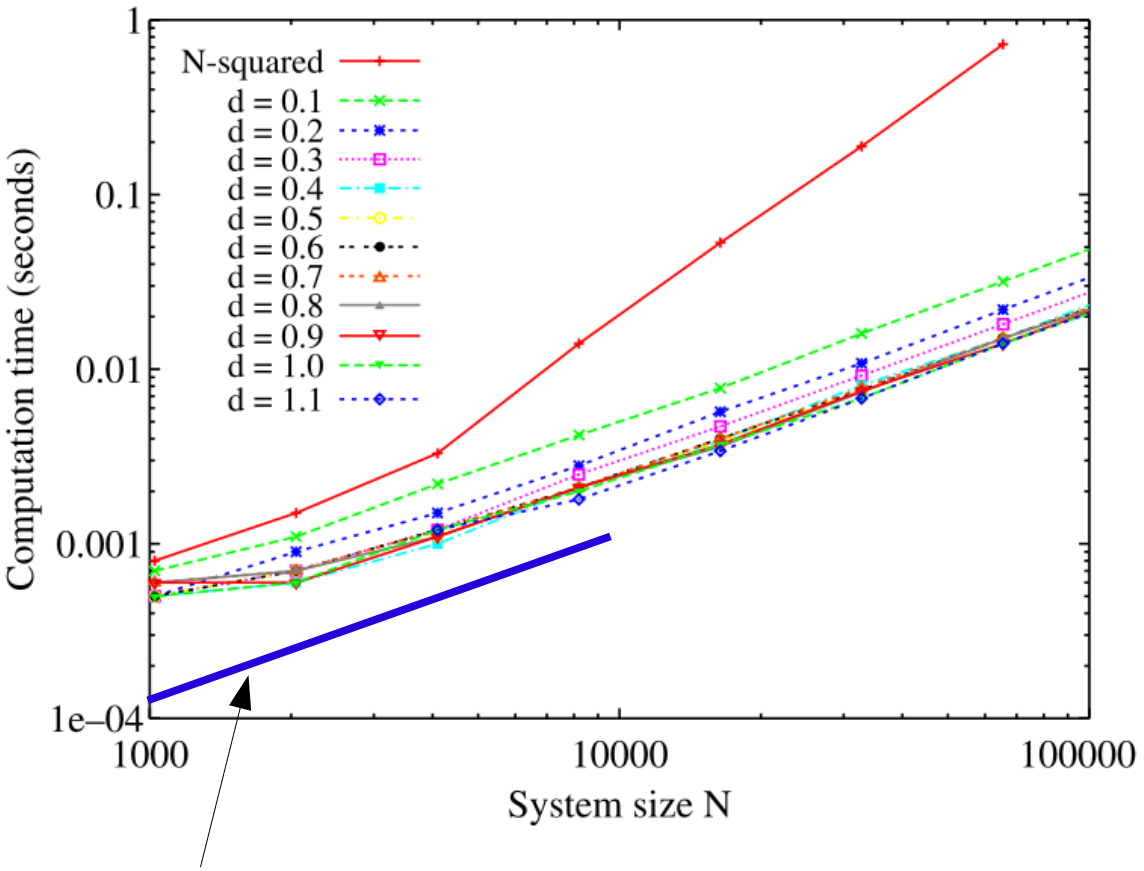
Cut-off

Easy to add other potentials

Templates controlling what gets calculated

Compared to the competition:

"Harvesting graphics power for MD simulations",
van Meel et al., Molecular Simulation, (2008):



Our program: Optimized for small N.

Speed-up compared to optimized CPU programs: ~ 20