

Molecular dynamics, on the GPU, using CUDA

Questions

- Why is energy stability important
- Mention an advantage and a disadvantage of classical molecular dynamics.
- How can one motivate the use of an interaction cutoff distance, physically and computationally

What is MD?

- We are theoretical physicists.
- MD are theoretical simulations aiming to describe and explain a multitude of phenomena.

Motivation

- Representative for many of the calculations we do
- Investigate the possibilities of using GPUs in our research codes
- Get an understanding of the maturity of the field

It works like this:

- Distribute atoms in your simulation box
- Give them initial velocities corresponding to an interesting temperature
- Have a good way of calculating the forces exerted on each atom
- Move the atoms in discrete steps
- Repeat the last few steps as many times you like.

Classical dynamics

- formulate the Lagrangian:

$$L = \sum_{i=1}^N L_i, \quad L_i(r_i, \dot{r}_i) = T(\dot{r}_i) + V(r_1, \dots, r_i, \dots, r_N)$$

- get momentum and Hamiltonian

$$p_i(r_i, \dot{r}_i, t) = \frac{\partial L}{\partial \dot{r}_i} \quad H = \sum_i \dot{r}_i p_i - L$$

- solve Hamilton's equations:

$$\dot{p}_i = -\frac{\partial H}{\partial r} \quad , \quad \dot{r}_i = -\frac{\partial H}{\partial p}$$

Potential

- The potential is the most complicated part. At any time it depends on the position of all other particles.
- Two families of solutions to this problem, classical and quantum.

Classical vs ab-initio

- Very fast
- Can handle systems up to 10^{10} atoms
- Potential dependent
- Wrong
- Slow
- 10^3 atoms if one has a really big computer
- Solves the Schrödinger/Dirac equations for the electrons. Results are very accurate.

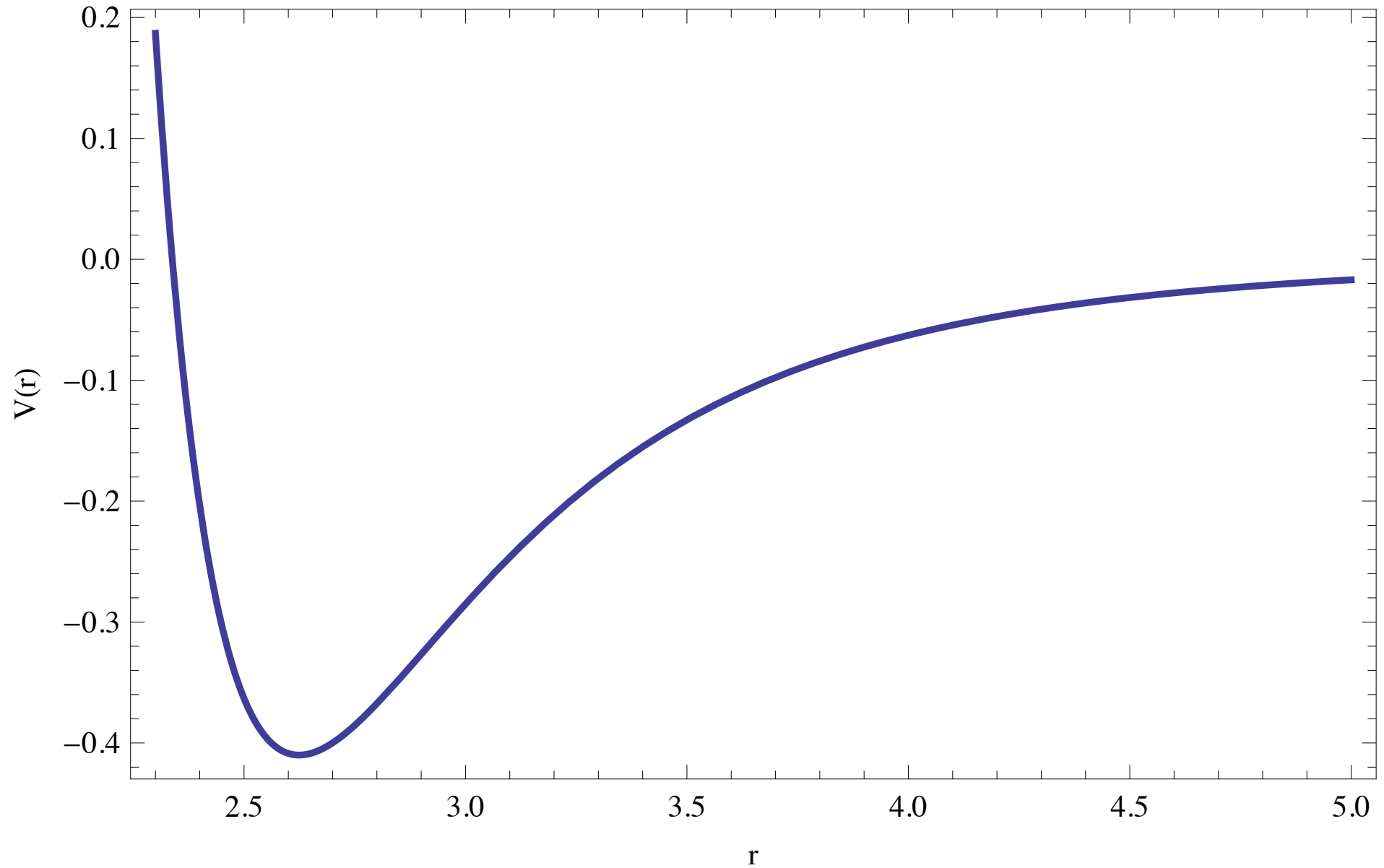
What did we choose

- Lennard-Jones classical pair potential, given by:

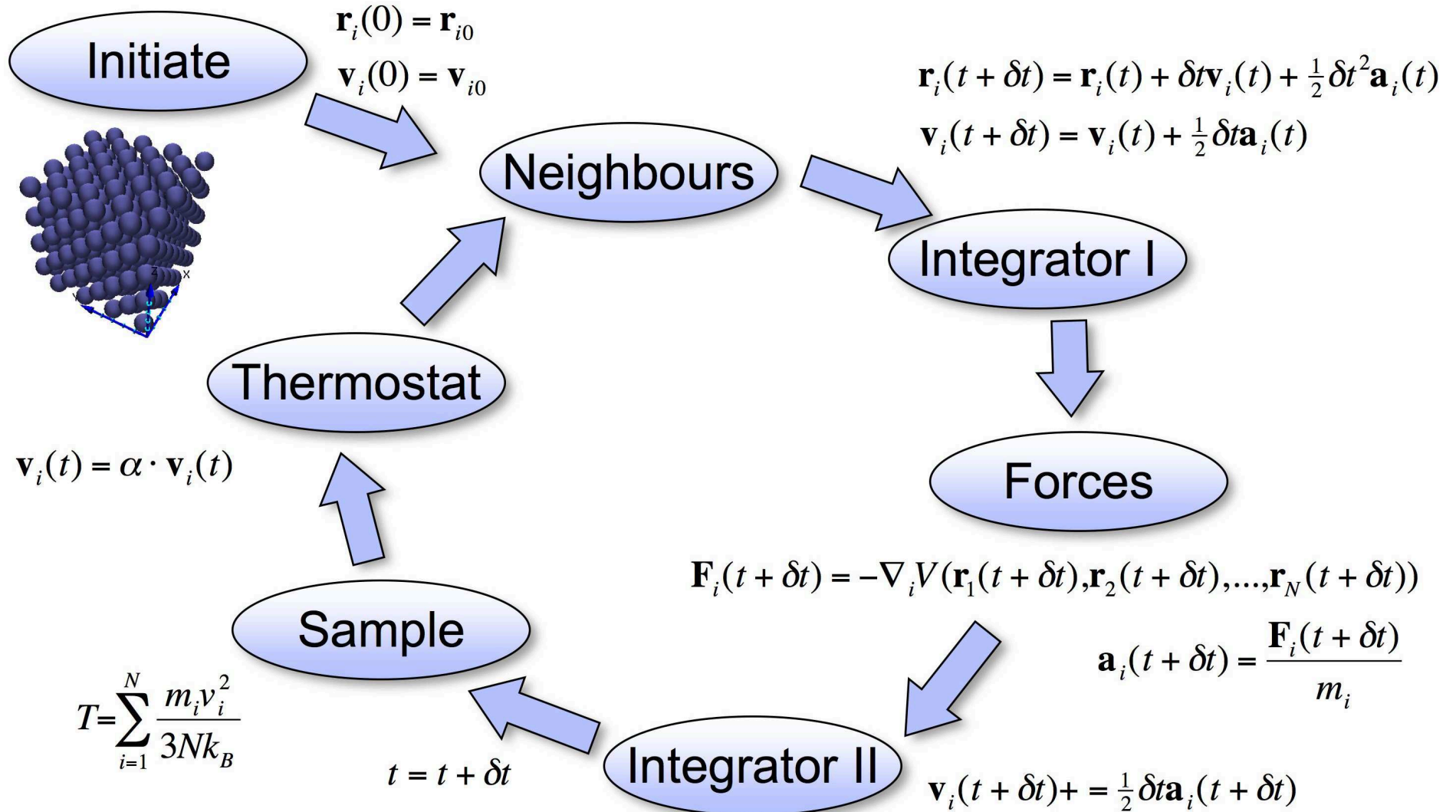
$$V(r_{ij}) = 4\epsilon \left(\frac{\sigma^{12}}{r_{ij}^{12}} - \frac{\sigma^6}{r_{ij}^6} \right)$$

- Only two parameters to fit to experimental results. Very easy to implement. Gives surprisingly good results considering its simplicity
- Impressed people in the 60's.

What does it look like?



The MD cycle



Our implementation 1.0

- We initialise the problem on the CPU, build the crystal lattice and set initial velocities.
- Move all the information to the GPU and keep it there for the duration
- Copy things back when we want to sample something.

Implementation

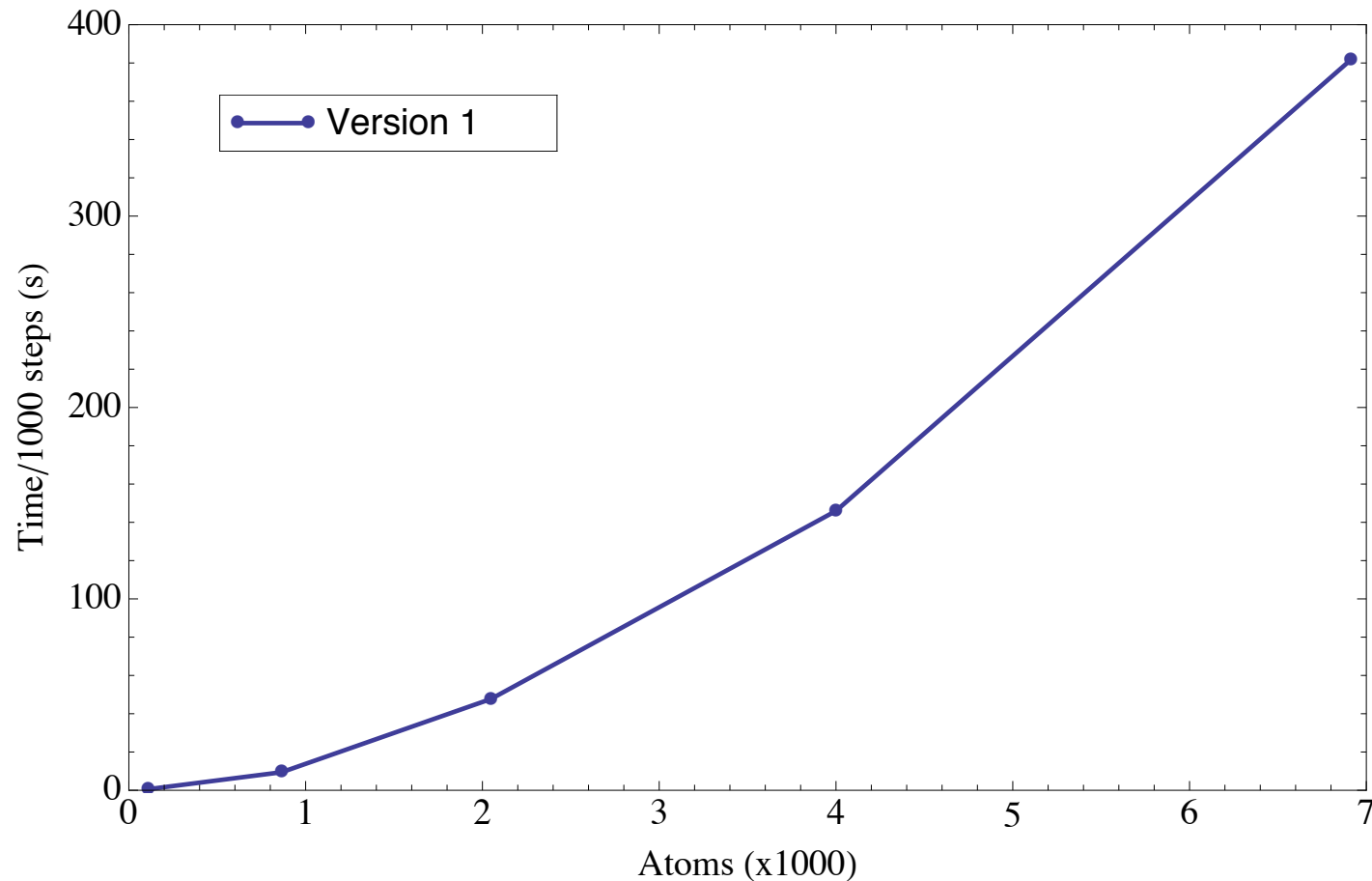
```
// main loop
for(j=0;j<nstep;j++){
    integrator1<<<dimGrid, dimBlock>>(cr,cv,cf,cb,cbi,a,ts,mass,npx,nty,npz);
    cudaThreadSynchronize();
    forces<<<dimGrid, dimBlock>>(cr,cv,cf,cb,cbi,a,ts,eps,sig,npx,nty,npz,cep);
    cudaThreadSynchronize();
    integrator2<<<dimGrid, dimBlock>>(cr,cv,cf,cb,cbi,a,ts,mass,npx,nty,npz,cek);
    cudaThreadSynchronize();

    if( j%printlog == 0 ){
        cudaMemcpy( lattice, cr, csize, cudaMemcpyDeviceToHost );
        cudaMemcpy( force, cf, csize, cudaMemcpyDeviceToHost );
        cudaMemcpy( epot, cep, np*sizeof(float), cudaMemcpyDeviceToHost );
        cudaMemcpy( ekin, cek, np*sizeof(float), cudaMemcpyDeviceToHost );
        cudaMemcpy( vel, cv, csize, cudaMemcpyDeviceToHost );

        dume=0.0;
        dumek=0.0;
        for(i=0;i<np;i++){
            dumek+=ekin[i];
            dume+=epot[i];
        }

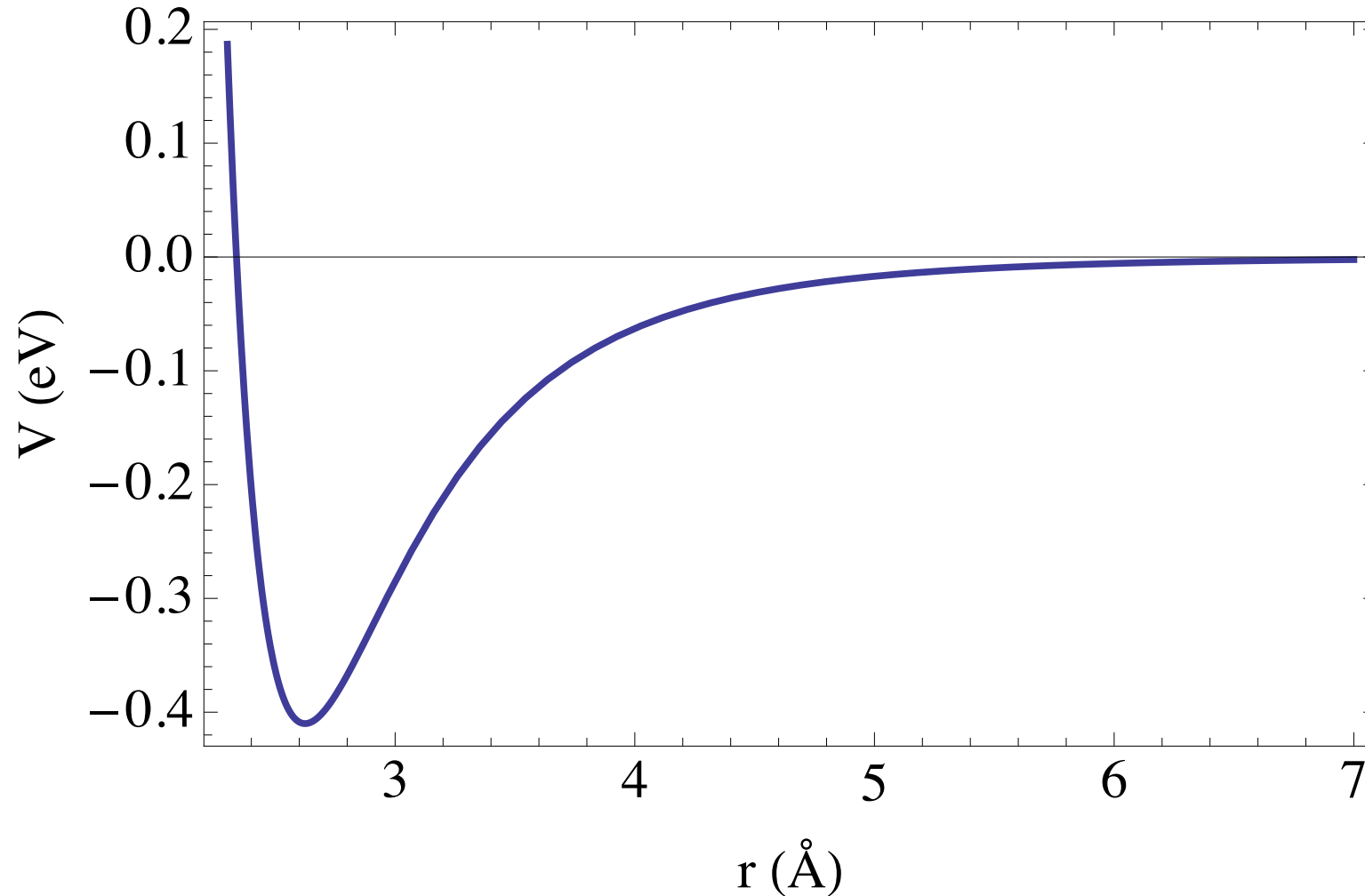
        printf("Force calc: %i\ttr= %f\ttf= %f\ttv= %f\ttep= %f\ttek= %f\ttet= %f\n"
            ,j,lattice[k],force[k],vel[k],dume/2,dumek,dume/2+dumek);
    }
}
```

Performance



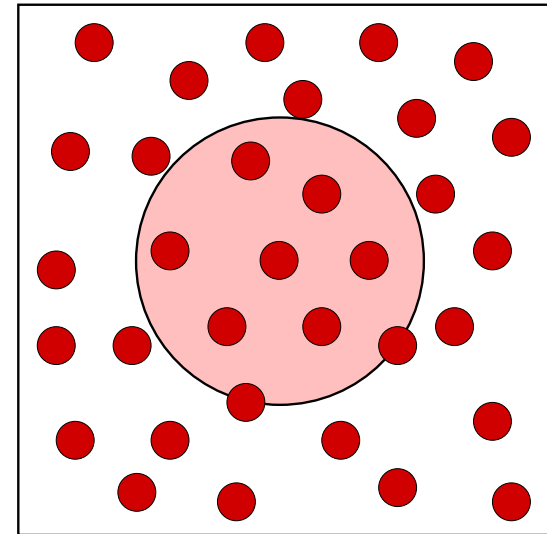
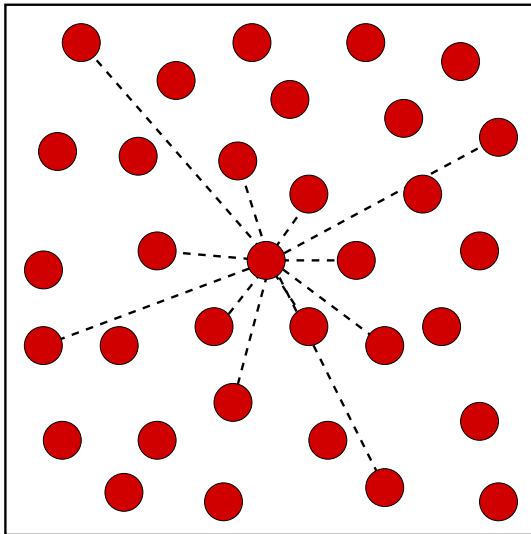
- Quadratic scaling with increasing number of particles. Not what we want.

Look at the potential again



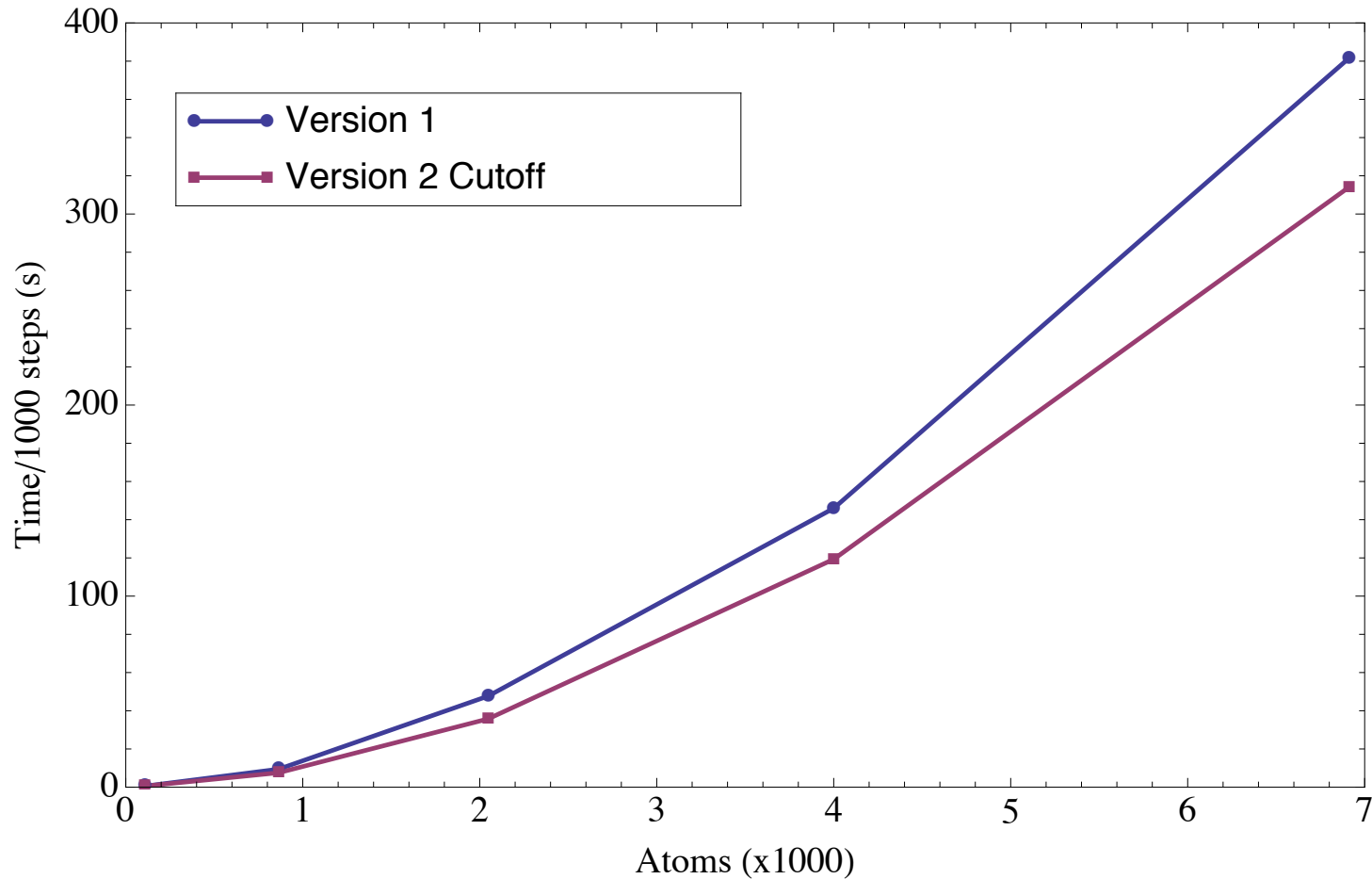
- It is probably ok to ignore interaction at large distances

Cutoff



Version 2

- Only evaluate forces if the distance is smaller than cutoff

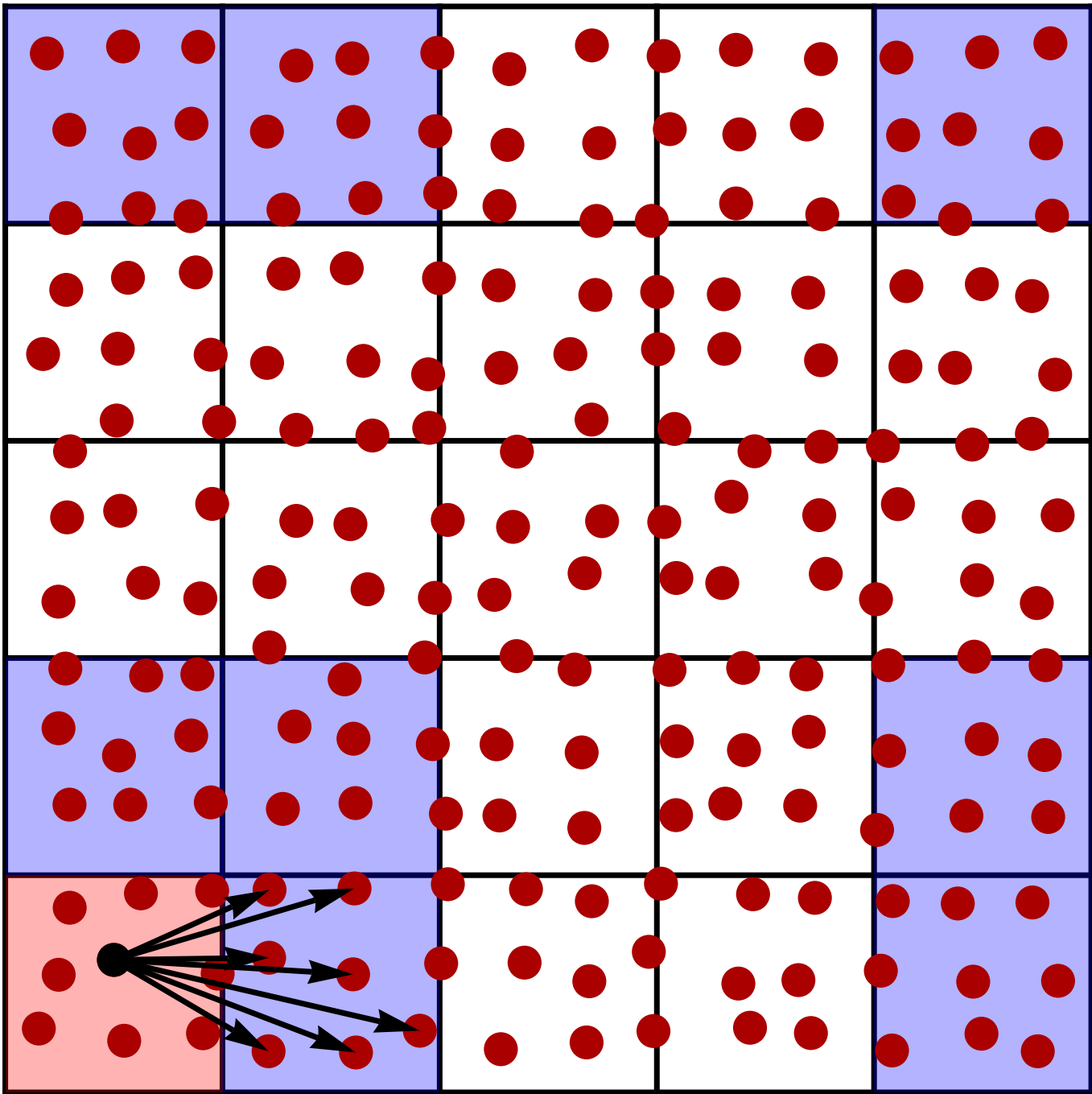


- We still need to calculate all the distances.

Version 3

Neighbourcells

- Divide the simulation box into cells of size slightly larger than the cutoff
- Only evaluate distances between neighbouring cells
- A bit of overhead in sorting and bookkeeping



Implementation

```
nbox=floor(sidelengthofsimulationbox/cutoff);
```

```
__device__ int devboxid(float x, float y, float z, float bz, int nbox){  
    int i,j,k,boxid;  
    i=floor(x/bz);  
    j=floor(y/bz);  
    k=floor(z/bz);  
    boxid=i+nbox*j+nbox*nbox*k;  
    return boxid;  
}
```

function that takes an atomic position and returns which box it belongs to

How many atoms in each box?

```
__global__ void sortera(float *r,int *boxcount, int *boxcontent,
                      float boxsize, int np, int atomsperbox, int nbox){
    const int atom = threadIdx.x + blockDim.x*blockIdx.x;
    if(atom>=np) return;
    // what box is the atom in?
    int bid = devboxid(r[atom*3+0],r[atom*3+1],r[atom*3+2],boxsize,nbox);
    // add one to the number of atoms in that box
    // atomicAdd returns the previous value to counter
    int counter = atomicAdd(&boxcount[bid], 1);
    // we have a maximum number of atoms that can reside in a box
    counter = min(counter, atomsperbox-1);
    // create a list containing the indices of the atoms, sorted according to which box
    // they live in
    boxcontent[bid*atomsperbox + counter] = atom;
}
```

The atomicAdd makes the addition thread safe.

Parallel model

- One block handles a cell
- Each thread handles an atom, and calculates the potential from all the atoms in its own and neighbouring cells

updated main loop

```
// set gpu grid
dim3 dimBlock(atomsperbox,1);
dim3 dimGrid(numberofboxes,1);
// main loop tjohoo!!!eleventy
for(j=0;j<nstep;j++){
    // first integrator
    integrator1<<<dimGridAtomic, dimBlockAtomic>>>(cr,cv,cf,cbi,np,ts,mass);
    cudaThreadSynchronize();

    // calculate forces
    forces<<< dimGrid, dimBlock, smsize >>>(cr,cv,cf,cboxnl,cboxcount,cboxcontent,
        cb,cbi,cutoff*cutoff,eps,sig,cep);
    cudaThreadSynchronize();

    // second integrator
    integrator2<<<dimGridAtomic, dimBlockAtomic>>>(cv,cf,np,ts,mass,cek);

    // reset the counter saying how many atoms are in each box
    resetcounter<<<dimGridReset, dimBlockReset>>>(cboxcount,nb);
    cudaThreadSynchronize();

    // sort the atoms into boxes again
    sortera<<<dimGridAtomic, dimBlockAtomic>>>(cr,cboxcount,cboxcontent,
        boxsized,np,atomsperbox,nbox);

    // copy and print
    if( j%printlog == 0 ){
        cudaMemcpy( lattice, cr, csize, cudaMemcpyDeviceToHost );
        cudaMemcpy( force, cf, csize, cudaMemcpyDeviceToHost );
        cudaMemcpy( epot, cep, np*sizeof(float), cudaMemcpyDeviceToHost );
        cudaMemcpy( ekin, cek, np*sizeof(float), cudaMemcpyDeviceToHost );
        cudaMemcpy( vel, cv, csize, cudaMemcpyDeviceToHost );
    }
}
```

first integrator

```
__global__ Former Tabell Diagram Kommentar iWork.com Mask Alfa Gruppera Avgruppera Overst Underst
void integrator1(float *r,float *v,float *f,float *bi, int np, float ts, float m){
    const int atom = threadIdx.x + blockDim.x*blockIdx.x;
    float acc[3], deltar[3],drc[3];
    float basi[9];
    int i,j;
    if(atom < np){
        // copy basis locally
        for(i=0;i<9;i++)
            basi[i]=bi[i];
        // calculate acceleration and displacement
        for(i=0;i<3;i++){
            acc[i]=f[atom*3+i]/(m*2);
            deltar[i]=ts*v[atom*3+i]+acc[i]*ts*ts;
        }
        // convert delta-r to direct coordinates
        for(i=0;i<3;i++) {
            drc[i]=0;
            for(j=0;j<3;j++)
                drc[i]=drc[i]+deltar[j]*basi[i*3+j];
        }
        // update positions
        for(i=0;i<3;i++){
            r[atom*3+i]=r[atom*3+i]+drc[i];
            // make sure the new positions obey PBC
            if ( r[atom*3+i] < 0.0f ) r[atom*3+i]+=1.0;
            if ( r[atom*3+i] > 1.0f ) r[atom*3+i]-=1.0;
        }
        // update velocities
        for(i=0;i<3;i++)
            v[atom*3+i]=v[atom*3+i]+acc[i]*ts;
    }
}
```

second integrator

```
__global__
void integrator2(float *v, float *f, int np, float ts, float m, float *ek){
    const int atom = threadIdx.x + blockDim.x*blockIdx.x;
    float acc[3];
    float normv;
    int i;
    if(atom < np){
        // get acceleration
        for(i=0;i<3;i++){
            acc[i]=f[atom*3+i]/(m*2);
        }
        // update velocities
        normv=0;
        for(i=0;i<3;i++){
            v[atom*3+i]=v[atom*3+i]+acc[i]*ts;
            // store the absolute value of the velocity squared
            normv+=v[atom*3+i]*v[atom*3+i];
        }
        // calculate kinetic energy
        ek[atom]=m*normv/2;
    }
}
```

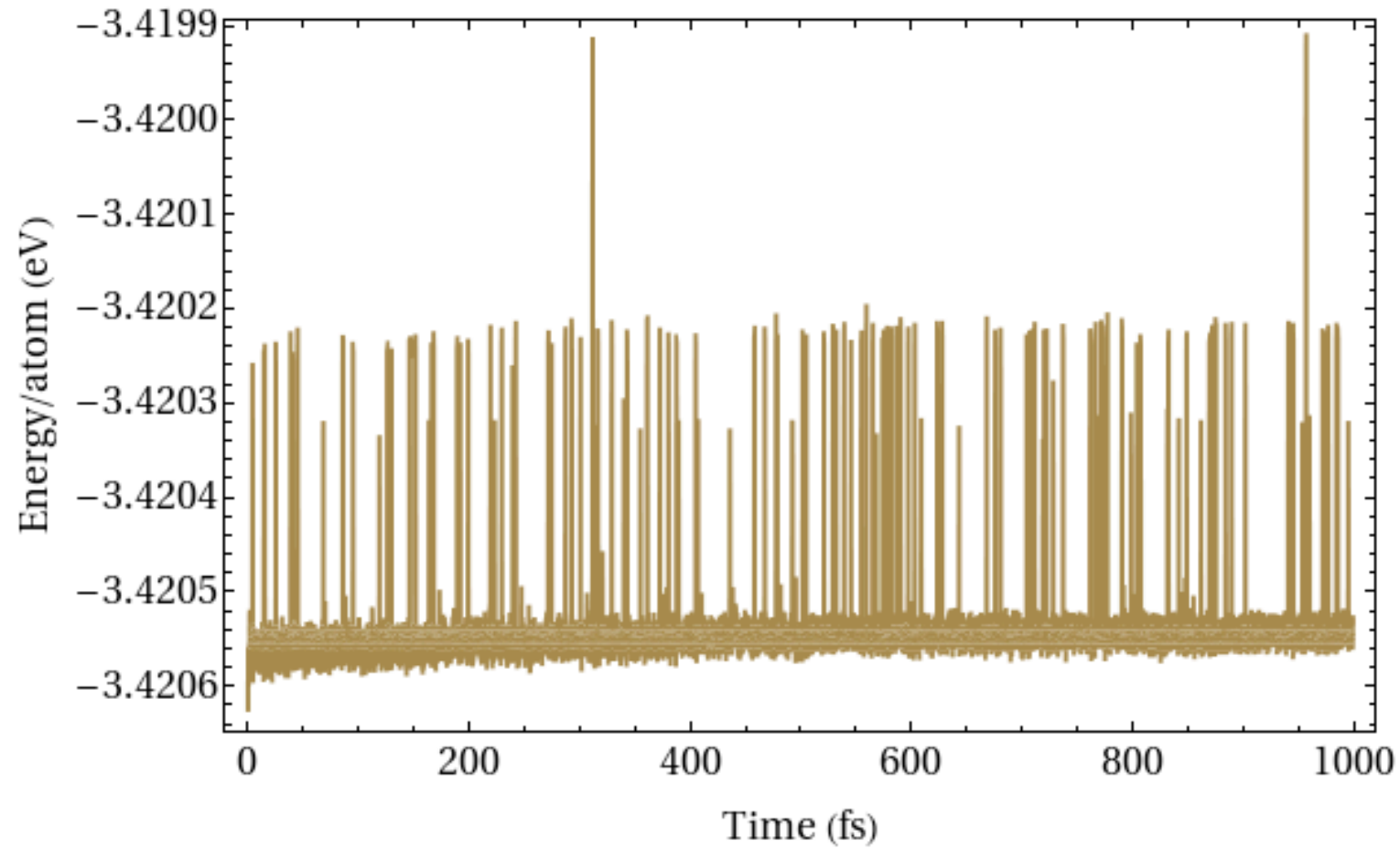
force calculations

```
// copy basis locally
for(i=0;i<9;i++)
    bas[i]=b[i];
// coordinate of current atom
for(j=0;j<3;j++) {
    dc1[j]=r[3*atom+j];
    force[j]=0.0f;
}
// store the number of atoms in the neighbouring cells locally
for(i=0;i<27;i++){
    boxidn = boxnl[boxid*27+i];
    lboxcount[i] = boxcount[boxidn];
}
// initialise potential energy
epot=0.0f;
// for each of the neighbouring cells (including the host cell)
// we store the atomic coordinates in a __shared__
for(i=0;i<27;i++){
    __syncthreads();
    if(latom < lboxcount[i]){
        boxidn = boxnl[boxid*27+i];
        nind = boxcontent[boxidn*apb + latom];
        refind[latom]=nind;
        rn[latom*3+0]=r[nind*3+0];
        rn[latom*3+1]=r[nind*3+1];
        rn[latom*3+2]=r[nind*3+2];
    }
    __syncthreads();
}
```

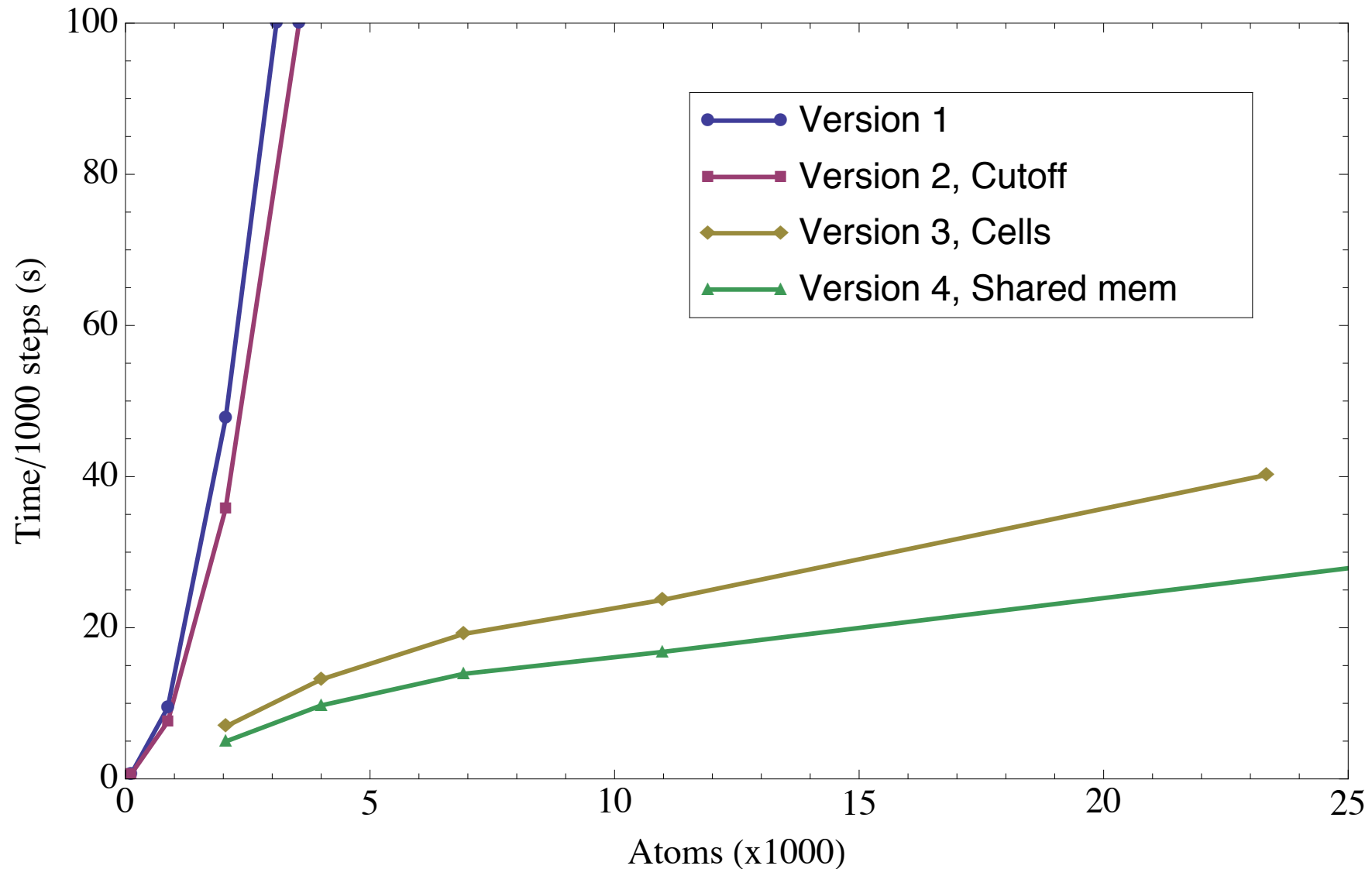
force calculations, cont

```
// for each of the atoms in the neighbouring cells
for(in=0;in<lboxcount[i];in++){
    // index of atom in the neighbouring cell
    nind = refind[in];
    // we can not have interaction with ourselves
    if( atom != nind ){
        // calculate r_ij vector in direct coordinates
        for(j=0;j<3;j++){
            dc2[j]=rn[3*in+j];
            // make sure the distance calculations take PBC into account
            if ( dc2[j]-dc1[j] <= -0.5f ) dc2[j]=dc2[j]+1.0;
            if ( dc2[j]-dc1[j] >  0.5f ) dc2[j]=dc2[j]-1.0;
            pvd[j]=dc2[j]-dc1[j];
        }
        // convert the pairvector to cartesian coordinates
        ...
        // check cutoff
        if(vn<cutoff2){
            // and finally calculate the force
            s6 = pow(sig*sig/vn,3);
            s12 = s6*s6;
            fn=4.0f*eps*(6.0f*s6/vn-12.0f*s12/vn);
            for(j=0;j<3;j++){
                force[j]=force[j]+pvc[j]*fn;
            }
            // and calculate the potential energy
            epot=epot+4*eps*(s12-s6);
        }
    }
}
```

Stability



It's faster!



Upper limit of 4000000 atoms (limited by memory)

Reflection

- The GPU is very fast
- Naive implementation is easy to do
- Optimized implementation become increasingly harder to debug
- Lack of easy debugging makes CUDA development much more frustrating
- A dedicated CUDA GPU would be nice

Questions?

