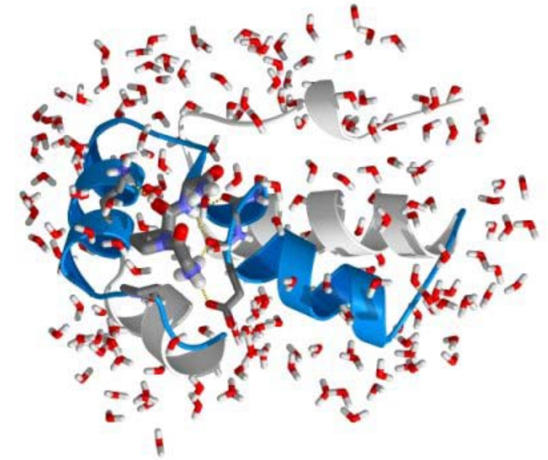
GROMACS
NVidia
Folding@home

Eric Darve

GROMACS: Erik Lindahl

- GROMACS provides *extremely high performance* compared to all other programs.
- Lot of algorithmic optimizations:
  - Own software routines to calculate the inverse square root.
  - Inner loops optimized to remove all conditionals.
  - Loops use SSE and 3DNow! multimedia instructions for x86 processors
  - For Power PC G4 and later processors: Altivec instructions provided
- normally 3-10 times faster than any other program.



Molecular Dynamics

- Simulate the dynamics of large molecules (like proteins,DNA) by solving Newton's equation of motion for the atoms.

$$\frac{d^2\, r_i}{dt^2} = \frac{F_i}{m_i}$$

- Equations of motion integrated in time to get position of all particles at later times.

Calculating the forces

- Several contributions to the force
  - Bond vibrations between atoms
  - Angle vibrations
  - Torsional/dihedral angles
  - Nonbonded electrostatics & Lennard-Jones between all atoms that are close in space
- The nonbonded interactions are VERY expensive!
  - N^2 problem if you include all atoms
  - Neighborlists/cutoffs make things better, but it’s still slow
  - We can calculate billions of nonbonded forces per second, but simulations can still take months to complete!

Nonbonded forces

- Accounts for 90-95% of the runtime in C/Fortran code
- Most common form:

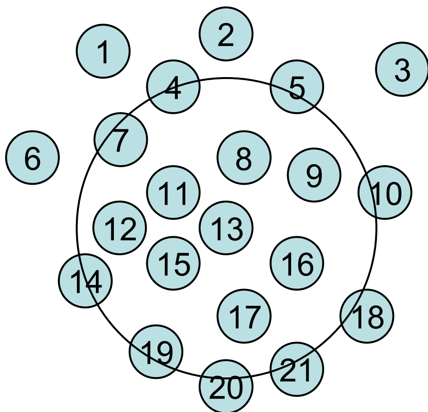
$$V_{nb} = \sum_{i,j} \left[\frac{1}{4\pi\epsilon_0} \frac{q_i \, q_j}{r_{ij}} + \left(\frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^{6}} \right) \right]$$

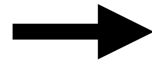
Electrostatics

Lennard-Jones

Using cutoffs & neighbor lists

- Neighbor list constructed every 10 steps.
- In practice: 10,000-100,000 atoms, with 100-200 neighbors in each list





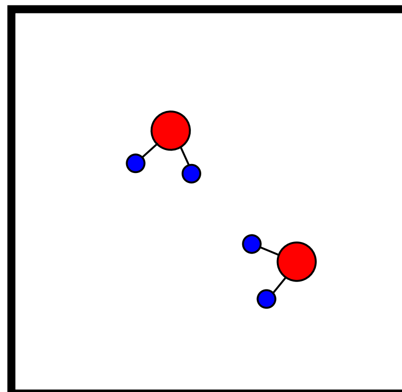
Neighbor list for atom 13 =
{ 8, 9, 11, 12, 15, 16, 17}

Optimizations

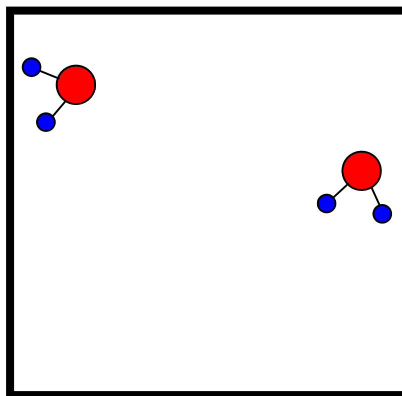
- Avoid all conditionals in the inner loops
- Improve cache usage.

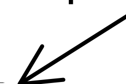
Avoid conditionals:

- Special loops for all possible interactions: LJ only, Coulomb only, LJ+Coulomb, etc.
- Periodic boundary conditions.
  - Use of different neighbor list depending on periodic shift that is applied



Molecule is shifted to the right
to account for periodicity of system





- Optimizing cache: Special loops for interactions between water molecules and other atoms, and yet another loop for water-water interactions.
  - By accessing data at the molecule level (rather than atom per atom) we make better use of cache.

What we do in the inner loop?

For each i atom {
- get i atom coordinates, charge & type
- get indices in neighborlist
- set temporary force variables (fx,fy,fz) to zero
- For each j atom in our neigborlist {
  - get j atom coordinates, charge & type
  - calculate vectorial distance; $\mathbf{dr} = \mathbf{r}_i - \mathbf{r}_j$
  - calculate $r^2=dx*dx+dy*dy+dz*dz$, and $1/r=1/sqrt(r^2)$
  - Calculate potential and vectorial force
  - Subtract the force from the j atom force
  - Add the force to our temporary force variables
- }
- Add the temporary force variables to the i atom forces
- Update the potential for the group atom i belongs to

}

Inner loop

Reading from memory

Costly

Writing to memory

Innermost loop in C

```
for (k=nj0;k<nj1;k++) {          //loop over indices in neighborlist
   jnr          = jjnr[k];       //get index of next j atom (array LOAD)
   j3           = 3*jnr;         //calc j atom index in coord & force arrays
   jx           = pos[j3];       //load x,y,z coordinates for j atom
   jy           = pos[j3+1];
   jz           = pos[j3+2];
   qq           = iq*charge[jnr]; //load j charge and calc. product
   dx           = ix - jx;       //calc vector distance i-j
   dy           = iy - jy;
   dz           = iz - jz;
   rsq          = dx*dx+dy*dy+dz*dz;       //calc square distance i-j
   rinv         = 1.0/sqrt(rsq);  //1/r
   rinvsq        = rinv*rinv;     //1/(r*r)
   vcoul         = qq*rinv;       //potential from this interaction
   fscal         = vcoul*rinvsq;  //scalarforce/|dr|
   vctot       += vcoul;          //add to temporary potential variable
   fix        += dx*fscal;        //add to i atom temporary force variable
   fiy        += dy*fscal;        //F=dr*scalarforce/|dr|
   fiz        += dz*fscal;
   force[j3]   -= dx*fscal;       //subtract from j atom forces
   force[j3+1]-= dy*fscal;
   force[j3+2]-= dz*fscal;
}
```

Normally, we use a table lookup and Newton-Raphson iteration instead of sqrt(x)

NVidia: NV30 & Cg

- GPU is a stream processor.
  - Processing units are programmable
- Characteristics of GPU:
  - optimized for 4-vector arithmetic
  - Cg has vector data types and operations
    e.g. float2, float3, float4
  - Cg also has matrix data types
    e.g. float3x3, float3x4, float4x4
- Some Math:
  - Sin/cos/etc.
  - Normalize
- Dot product: `dot(v1,v2);`
- Matrix multiply:
  - matrix-vector: `mul(M, v);` // returns a vector
  - vector-matrix: `mul(v, M);` // returns a vector
  - matrix-matrix: `mul(M, N);` // returns a matrix

Assembly or High-level?

Assembly

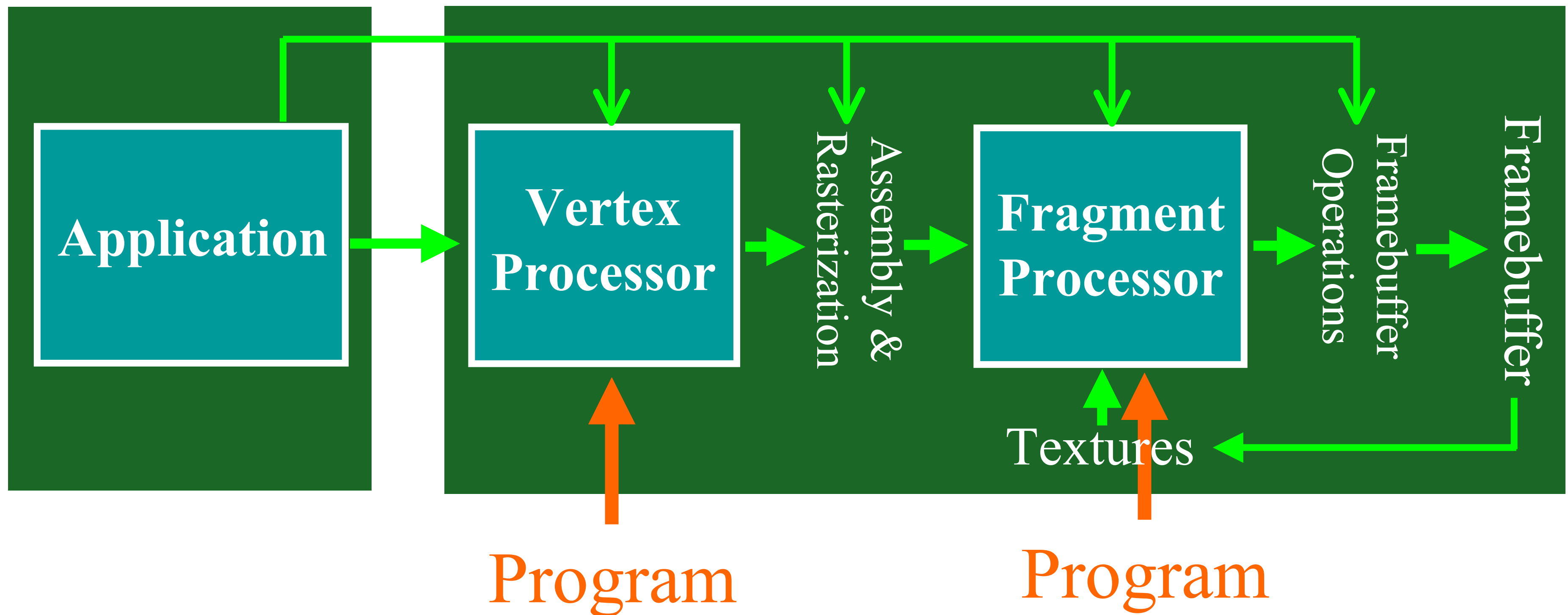
```
…
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

or

Cg

```
COLOR cPlastic = Ca + Cd * dot(Nf, L)
    + Cs * pow(max(0, dot(Nf, H)), phongExp);
```

Cg uses separate vertex and fragment programs



inner loop in Cg: Ian Buck

```
/* Find the index and coordinates of j atom */
jnr = f4tex1D (jjnr, k);

/* Get the atom position */
j1 = f3tex1D(pos, jnr.x);
j2 = f3tex1D(pos, jnr.y);
j3 = f3tex1D(pos, jnr.z);
j4 = f3tex1D(pos, jnr.w);
```

We compute four interactions at a time so that we can take advantage of high performance for 4-vector arithmetic.

```
/* Get the vectorial distance, and r^2 */
d1 = i - j1;
d2 = i - j2;
d3 = i - j3;
d4 = i - j4;

rsq.x = dot(d1, d1);
rsq.y = dot(d2, d2);
rsq.z = dot(d3, d3);
rsq.w = dot(d4, d4);

/* Calculate 1/r */
rinv.x = rsqrt(rsq.x);
rinv.y = rsqrt(rsq.y);
rinv.z = rsqrt(rsq.z);
rinv.w = rsqrt(rsq.w);
```

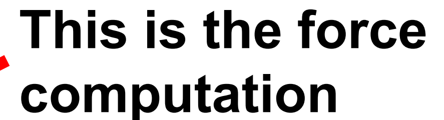
```
/* Calculate Interactions */
rinvsq      = rinv * rinv;
rinvsix     = rinvsq * rinvsq * rinvsq;

vnb6        = rinvsix * temp_nbfp;
vnb12       = rinvsix * rinvsix * temp_nbfp;
vnbtot      = vnb12 - vnb6;

qq          = iqA * temp_charge;
vcoul       = qq*rinv;
fs          = (12f * vnb12 - 6f * vnb6 + vcoul) * rinvsq;
vctot       = vcoul;

/* Calculate vectorial force and update local i atom force */
fi1         = d1 * fs.x;
fi2         = d2 * fs.y;
fi3         = d3 * fs.z;
fi4         = d4 * fs.w;
```

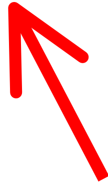
Highly efficient float4 arithmetic



This is the force computation

Computing total force due to 4 interactions

```
ret_prev.fi_with_vtot.xyz += fi1 + fi2 + fi3 + fi4;
ret_prev.fi_with_vtot.w   += dot(vnbtot, float4(1, 1, 1, 1))
                                 + dot(vctot, float4(1, 1, 1, 1));
```



Computing total potential energy for this particle

Return type is:

```
struct inner_ret {
  float4 fi_with_vtot;
};
```

Contains x, y and z coordinates of force and total energy.

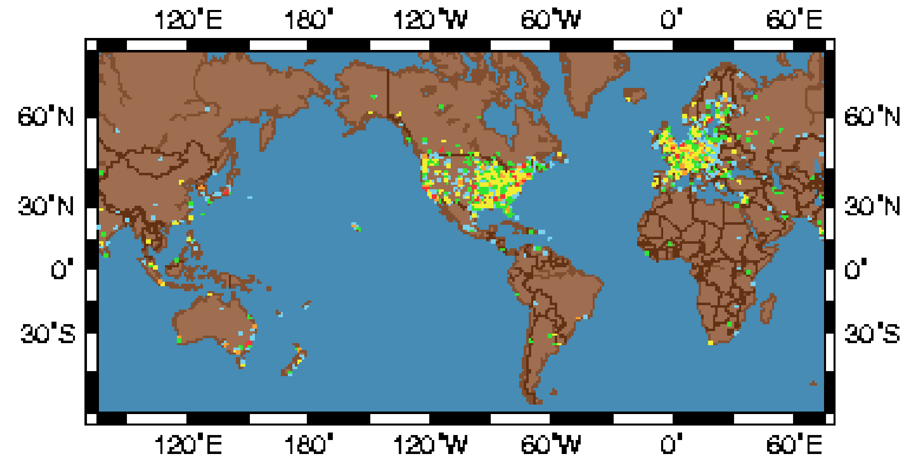
Technical points of implementation

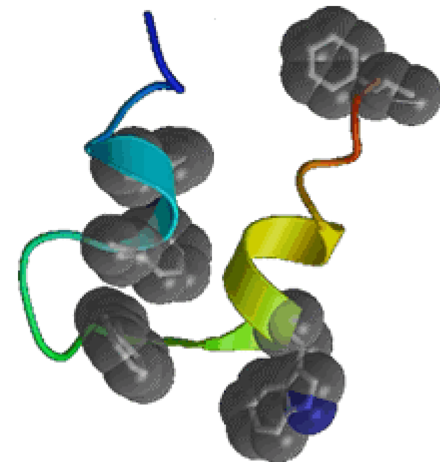
- We do not update the forces on the j atoms in the innermost loop: only the force on i atom is updated.
  - Update force on j seems complicated (perhaps possible nevertheless)
  - We have both atom i in atom j's neighbor list, AND atom j in atom i's neighbor list
    (i.e. calculate the interaction twice).
- We manipulate the neighbor lists to get chunks of equal length (say 16 j particles) since the graphics cards can't do general for-loops.
  - "for" and "while" loops need to be explicitely unrollable.
- The bandwidth necessary to read back the forces from the graphics card to the main CPU is in the order of 1/10th of the available GPU-CPU bandwidth.

Folding@home: Vijay Pande

What does Folding@Home do?

Folding@Home is a distributed computing project which studies protein folding, misfolding, aggregation, and related diseases. We use novel computational methods and large scale distributed computing, to simulate timescales thousands to millions of times longer than previously achieved. This has allowed us to simulate folding for the first time, and to now direct our approach to examine folding related disease.





Results from Folding@Home simulations of villin