GPU Teaching Kit

Accelerated Computing

Module 16 - Application Case Study – Electrostatic Potential Calculation

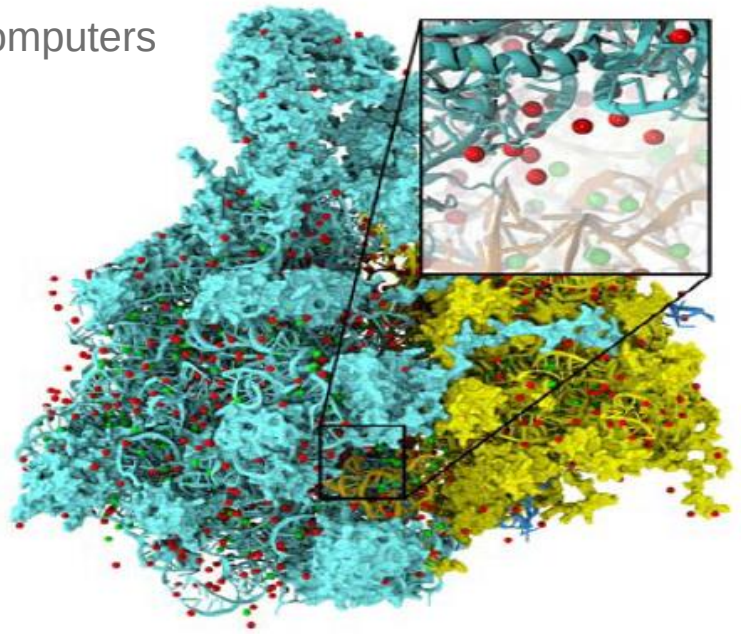
Lecture 16.1 - Electrostatic Potential Calculation

Objective

- To learn how to apply parallel programming techniques to an application
  - Thread coarsening for more work efficiency
  - Data structure padding for reduced divergence
  - Memory access locality and pre-computation techniques

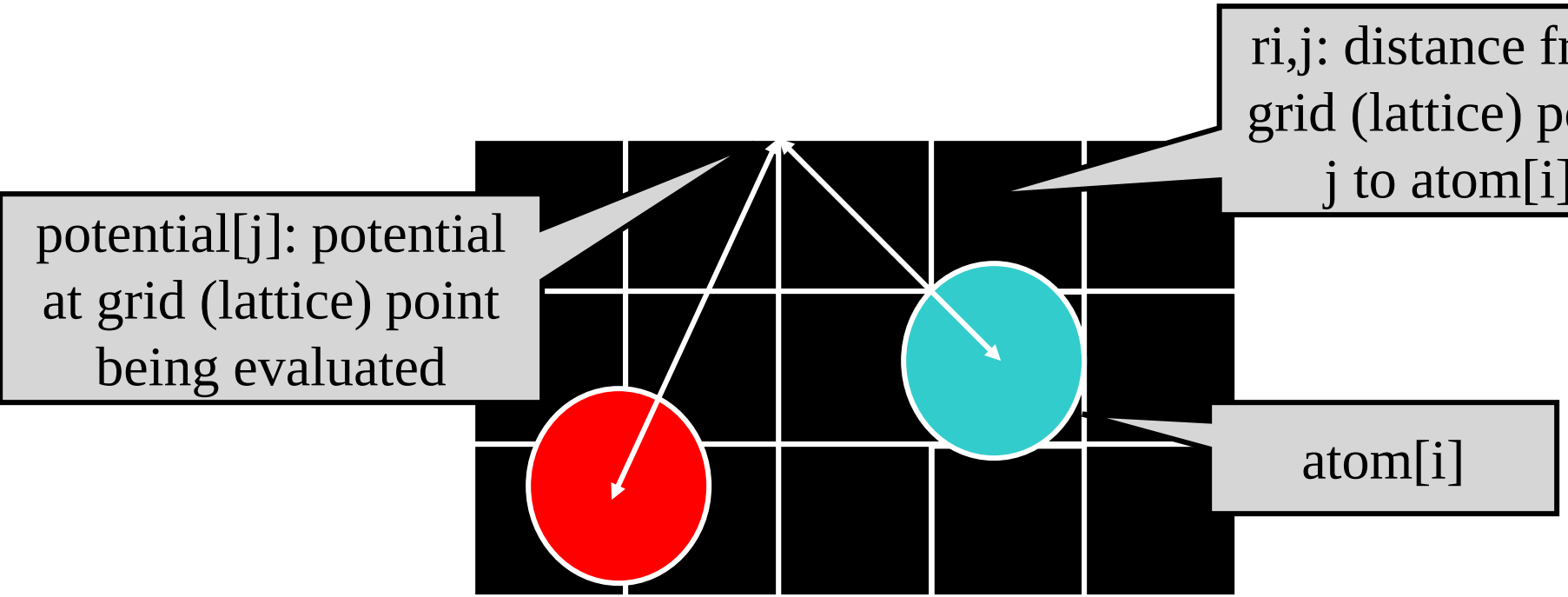
VMD

- Visual Molecular Dynamics
  - Visualizing, animating, and analyzing bio-molecular systems
  - More than 200,000 users worldwide
  - Batch (movie making) vs. interactive mode
  - Runs on laptops, desktops, clusters, supercomputers



Electrostatic Potential Map

- Calculate initial electrostatic potential map around the simulated structure considering the contributions of all atoms
  - Most time consuming, focus of our example.



Electrostatic Potential Calculation

- The contribution of atom[i] to the electrostatic potential at lattice[j] is potential[j] = atom[i].charge / rij.
- In the Direct Coulomb Summation method, the total potential at lattice point j is the sum of contributions from all atoms in the system.

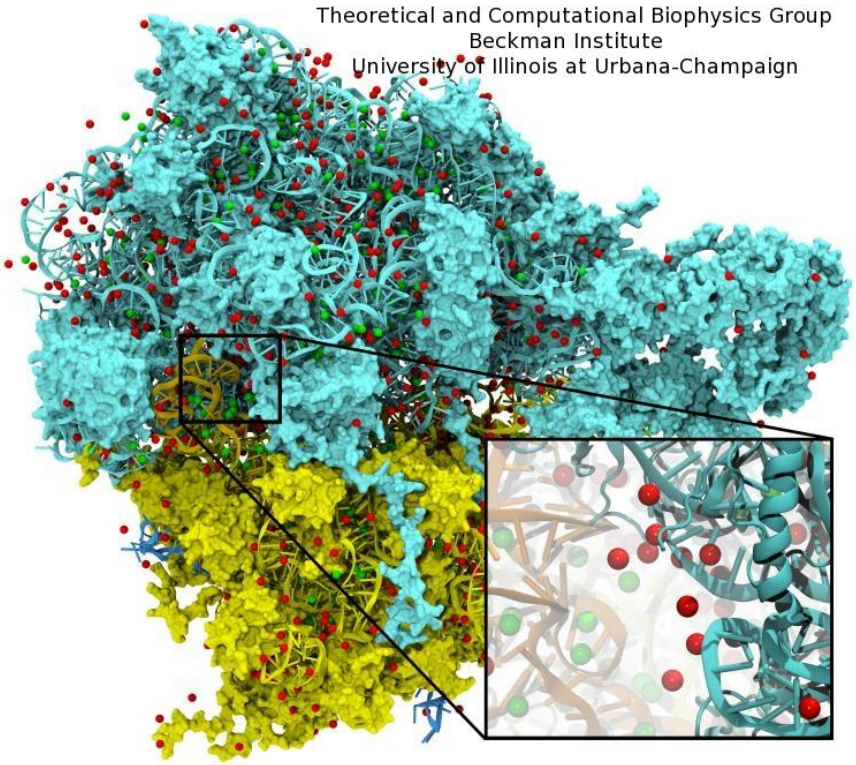
Overview of Direct Coulomb Summation (DCS) Algorithm

- One way to compute the electrostatic potentials on a grid, ideally suited for the GPU
  - All atoms affect all map lattice points, most accurate
- For each lattice point, sum potential contributions for all atoms in the simulated structure:

  potential += charge[i] / (distance to atom[i])
- Approximation-based methods such as cut-off summation can achieve much higher performance at the cost of some numerical accuracy and flexibility

Irregular Input vs. Regular Output

- Atoms come from modeled molecular structures, solvent (water) and ions
  - Irregular by necessity
- Energy grid models the electrostatic potential value at regularly spaced points
  - Regular by design



Summary of Sequential C Version

- Algorithm is input oriented
  - For each input atom, calculate its contribution to all grid points in an x-y slice
- Output (energy grid) is regular
  - Simple linear mapping between grid point indices and modeled physical coordinates
- Input (atom) is irregular
  - Modeled x,y,z coordinate of each atom needs to be stored in the atom array
- The algorithm is efficient in performing minimal number of calculations on distances, coordinates, etc.

8

A Sequential C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms,
             int numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {    // calculate potential contribution of each atom
        float dx = x - atoms[n  ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
  }
}
```

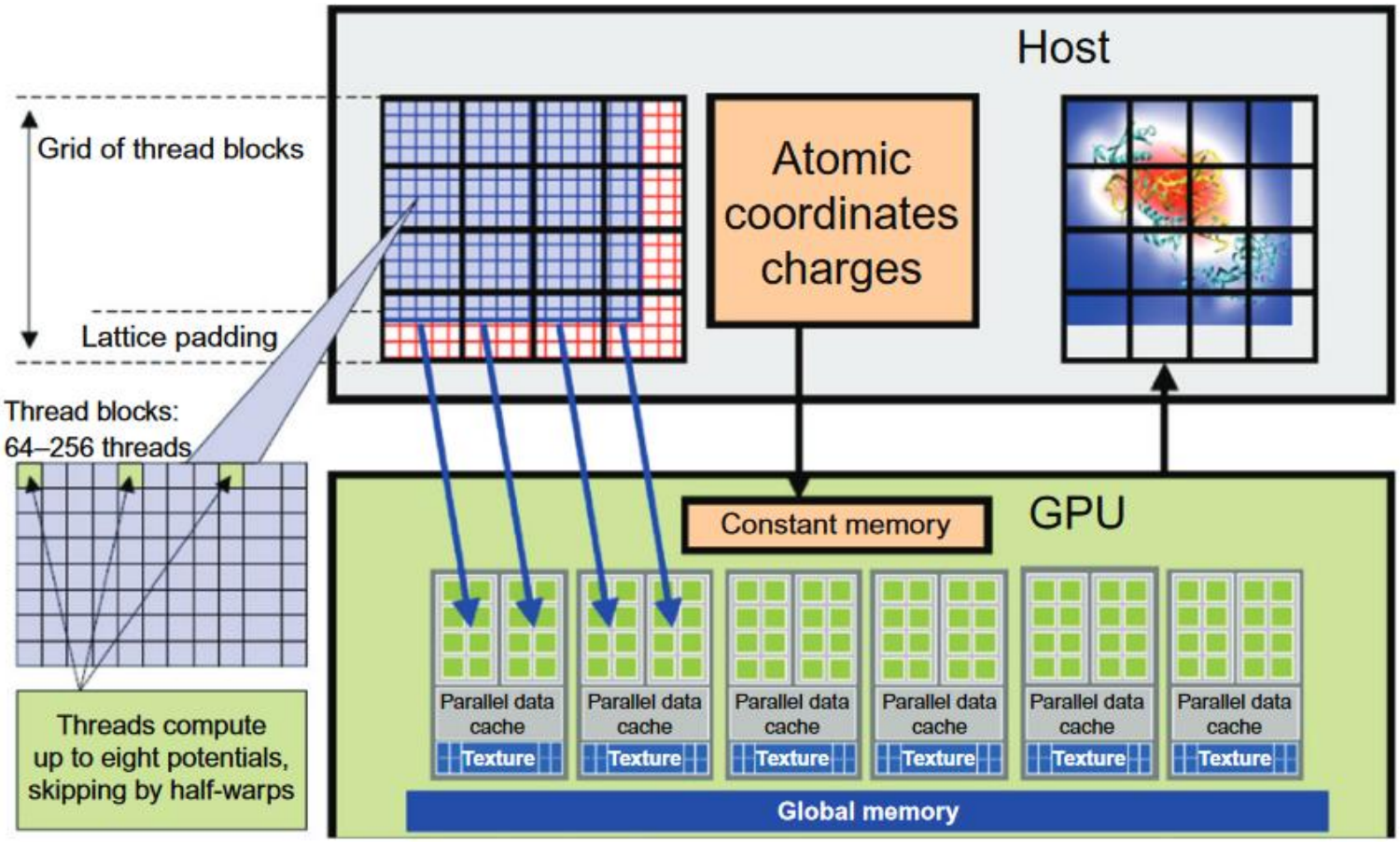
CUDA DCS Implementation Overview

- Allocate and initialize potential map memory on host CPU
- Allocate potential map slice buffer on GPU
- Preprocess atom coordinates and charges
- Loop over potential map slices:
  - Copy potential map slice from host to GPU
  - Loop over groups of atoms:
    - Copy atom data to GPU
    - Run CUDA Kernel on atoms and potential map slice on GPU
  - Copy potential map slice from GPU to host
- Free resources

CUDA Parallelization

- Operations done fore each lattice point are independent of each other, what makes them good candidate for parallelization
- Atom positions and charges are used for all lattice point calculation, retriving data is apotential bottleneck.

Overview of the DCS kernel design



A Fast DCS CUDA Gather Kernel

```
void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing, float z, float *atoms,
int numatoms) {

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int atomarrdim = numatoms * 4;
  int k = z / gridspacing;
  float y = gridspacing * (float) j;
  float x = gridspacing * (float) i;
  float energy = 0.0f;
  for (int n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each atom
        float dx = x - atoms[n    ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
   energygrid[grid.x*grid.y*k + grid.x*j + i] += energy;
}
```

A Fast DCS CUDA Gather Kernel

```
...
float curenergy = energygrid[outaddr];
float coorx = gridspacing * xindex;
float coory = gridspacing * yindex;
int atomid;
float energyval=0.0f;
 for (atomid=0; atomid<numatoms; atomid++) {
  float dx = coorx - atominfo[atomid].x;
  float dy = coory - atominfo[atomid].y;
  energyval += atominfo[atomid].w *
                        rsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
}
energygrid[outaddr] = curenergy + energyval;
```

Start global memory reads early. Kernel hides some of its own latency.

Only dependency on global memory read is at the end of the kernel…

Adjustments

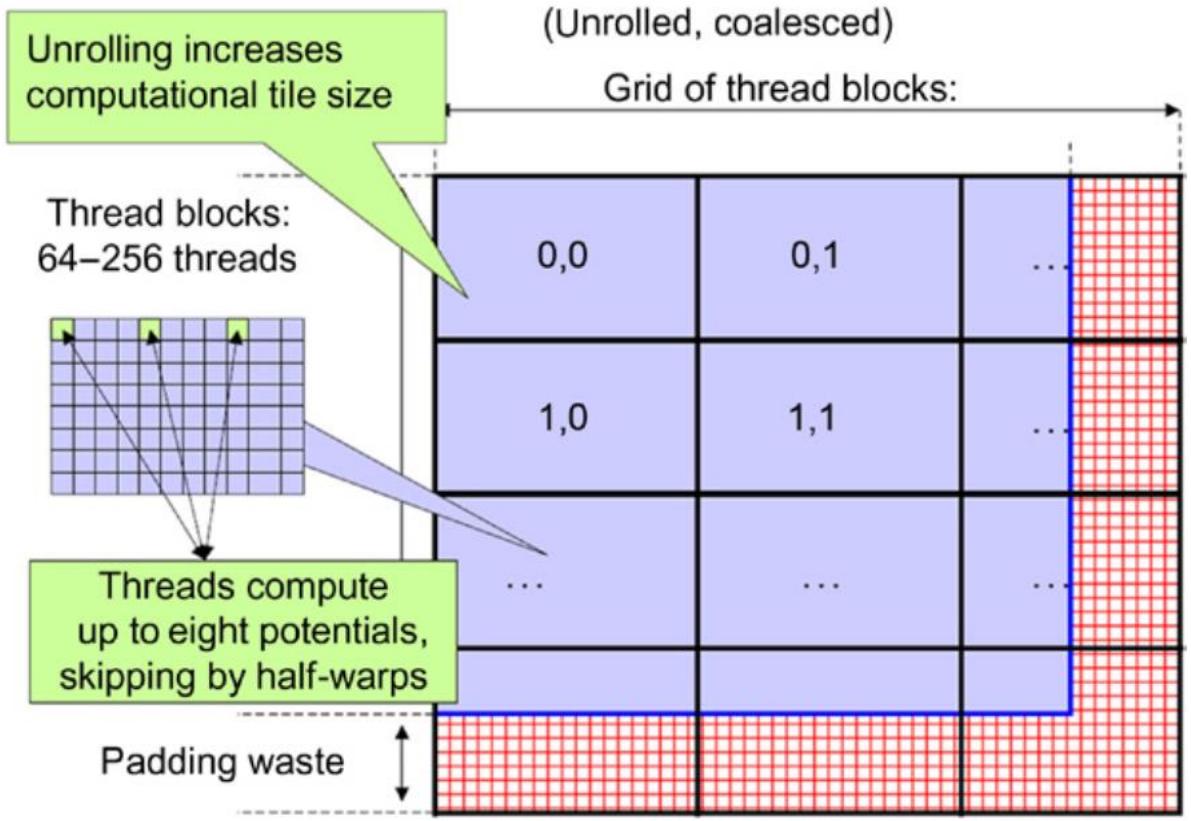
- Ratio of calculation to memory acess ratio is suboptimal
- dz*dz is always the same and can be cached

A Fast DCS CUDA Gather Kernel v2

```
…for (atomid=0; atomid<numatoms; atomid++) {
  float dy = coory - atominfo[atomid].y;
  float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
  float x = atominfo[atomid].x;
  float dx1 = coorx1 - x;
  float dx2 = coorx2 - x;
  float dx3 = coorx3 - x;
  float dx4 = coorx4 - x;
  float charge = atominfo[atomid].w;
  energyvalx1 += charge * rsqrtf(dx1*dx1 + dysqpdzsq);
  energyvalx2 += charge * rsqrtf(dx2*dx2 + dysqpdzsq);
  energyvalx3 += charge * rsqrtf(dx3*dx3 + dysqpdzsq);
  energyvalx4 += charge * rsqrtf(dx4*dx4 + dysqpdzsq);
}
```

Compared to non-unrolled kernel: memory loads are decreased by 4x, and FLOPS per evaluation are reduced, but register use is increased…

Memory Coalesing



A Fast DCS CUDA Gather Kernel v2

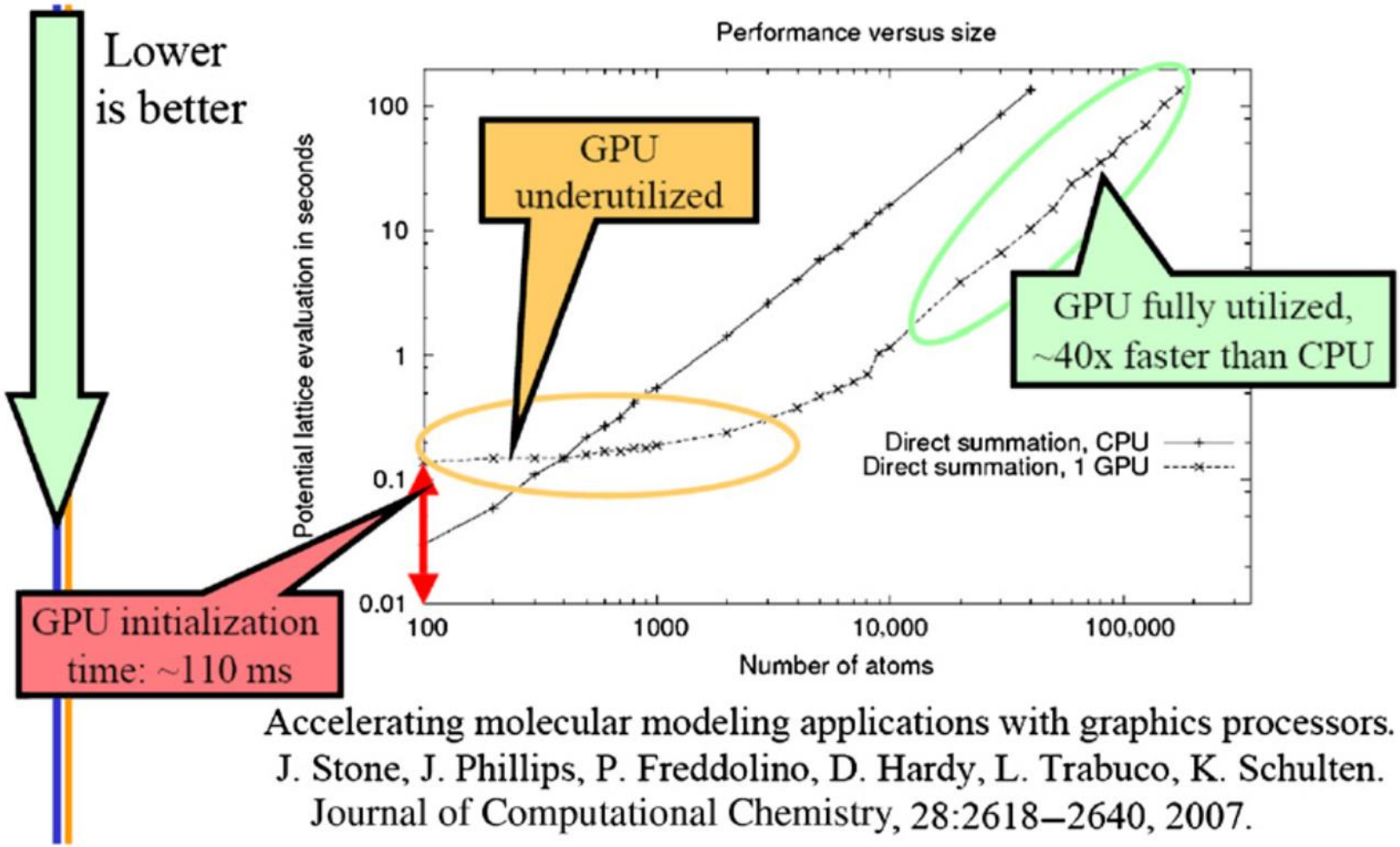
```
…float coory = gridspacing * yindex;
  float coorx = gridspacing * xindex;
  float gridspacing_coalesce = gridspacing * BLOCKSIZEX;
  int atomid;
  for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx - atominfo[atomid].x;
[…]
    float dx8 = dx7 + gridspacing_coalesce;
    energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
[…]
    energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
  }
  energygrid[outaddr                    ] += energyvalx1;
[...]
  energygrid[outaddr+7*BLOCKSIZEX] += energyvalx7;
```

Points spaced for memory coalescing

Reuse partial distance components dy^2 + dz^2

Global memory ops occur only at the end of the kernel, decreases register use

Speed comparison



Accelerating molecular modeling applications with graphics processors.
J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten.
Journal of Computational Chemistry, 28:2618–2640, 2007.

GPU Teaching Kit

Accelerated Computing