Real Application Example: Molecular Visualization and Analysis

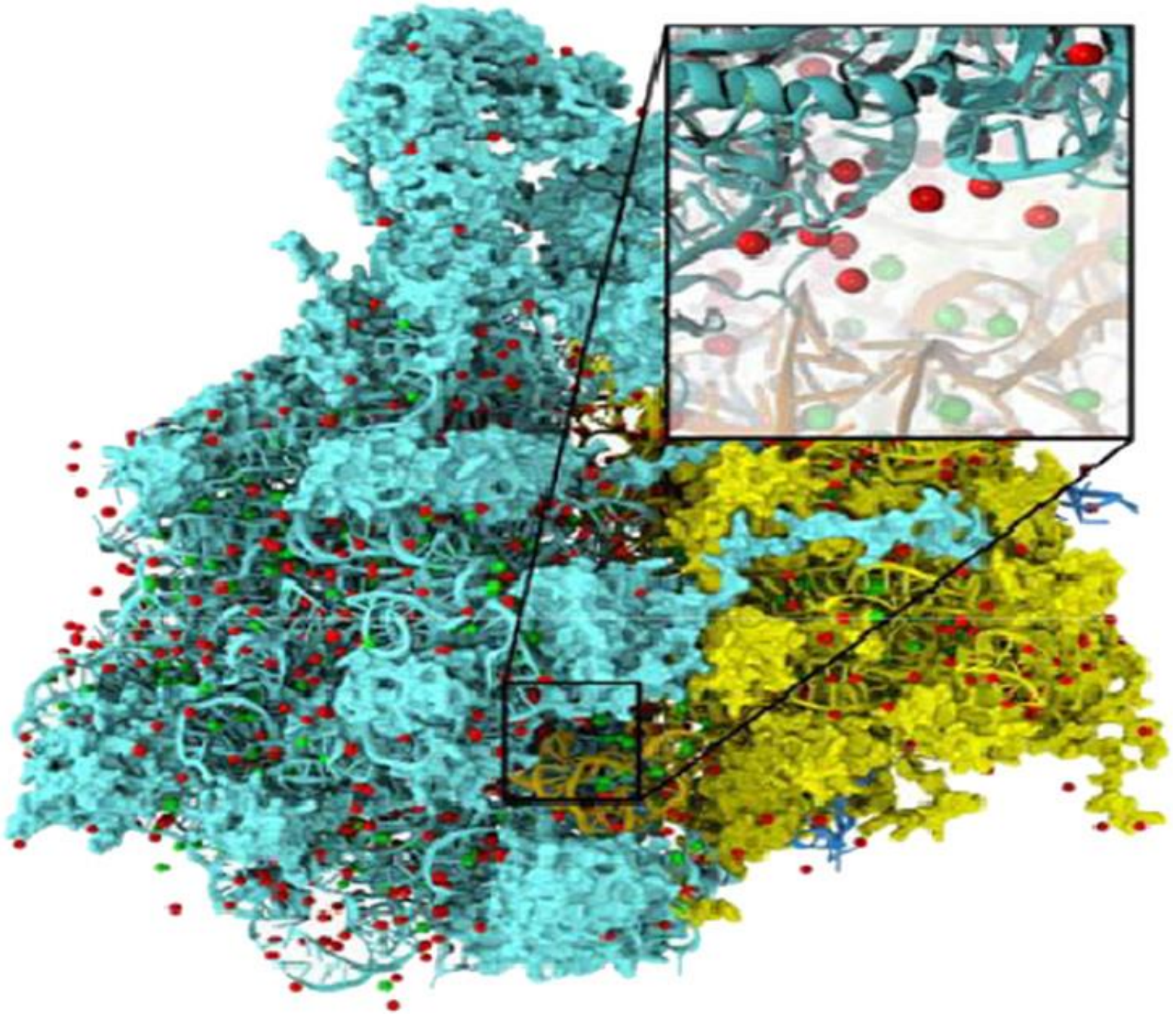
Ahmed El-Mahdy

Objectives

- Illustrate the process of computational thinking in parallelizing an application
- Provides for a challenging real application scenario

Visual Molecula Dynamics (VMD

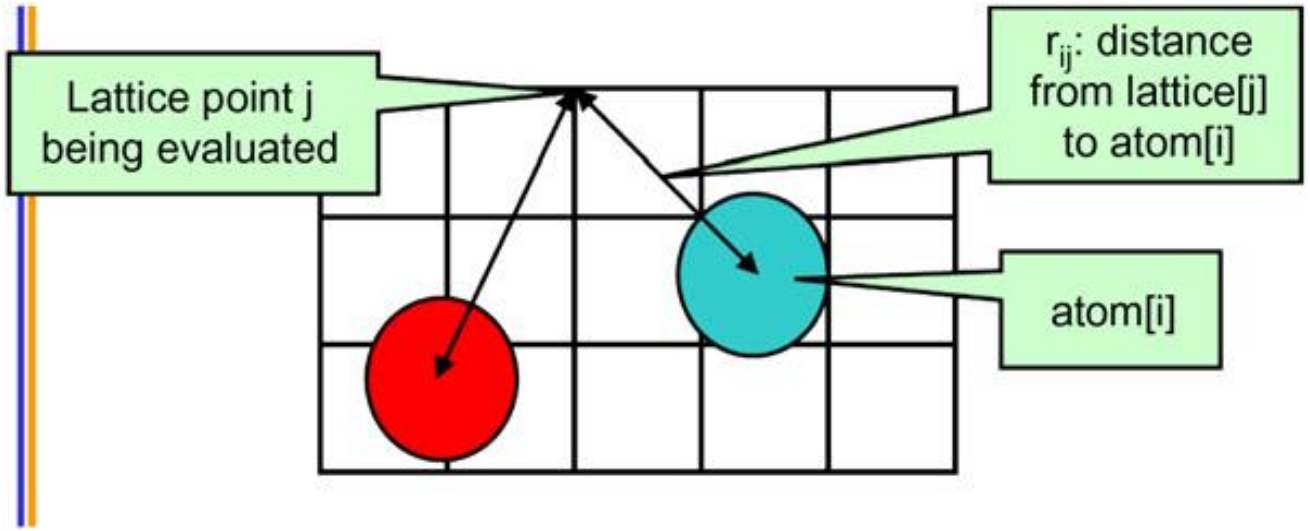
- Computes electrostatic potential values at spatial grid points of a molecular system
- Visualising large data sets in sequencing data, quantum chemistry, and volumic data



Electrostatic Potential Map

- Identify locations where ions (round dots around the large molecules) can fit
- Compute time-averaged potentials during molecular dynamics simulation, for visualisation and analysis of simulation results
- The computation is done using Direct Coulomb Summation (DCS)

Direct Coulomb Summation (DCS)



The contribution of `atom[i]` to the electrostatic potential at lattice point *j* (`potential[j]`) is `atom[i].charge`/r_{ij}. In the DCS method, the total potential at lattice point *j* is the sum of contributions from all atoms in the system.

Kernel Implementation for computing a 2D slide of a 3D grid

Base Coulomb Potential Calculation Code for a 2D Slice

```
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;
    }
  }
}
```

Observations

- Computation for each point is independent from others -> massively parallel

Problem decomposition

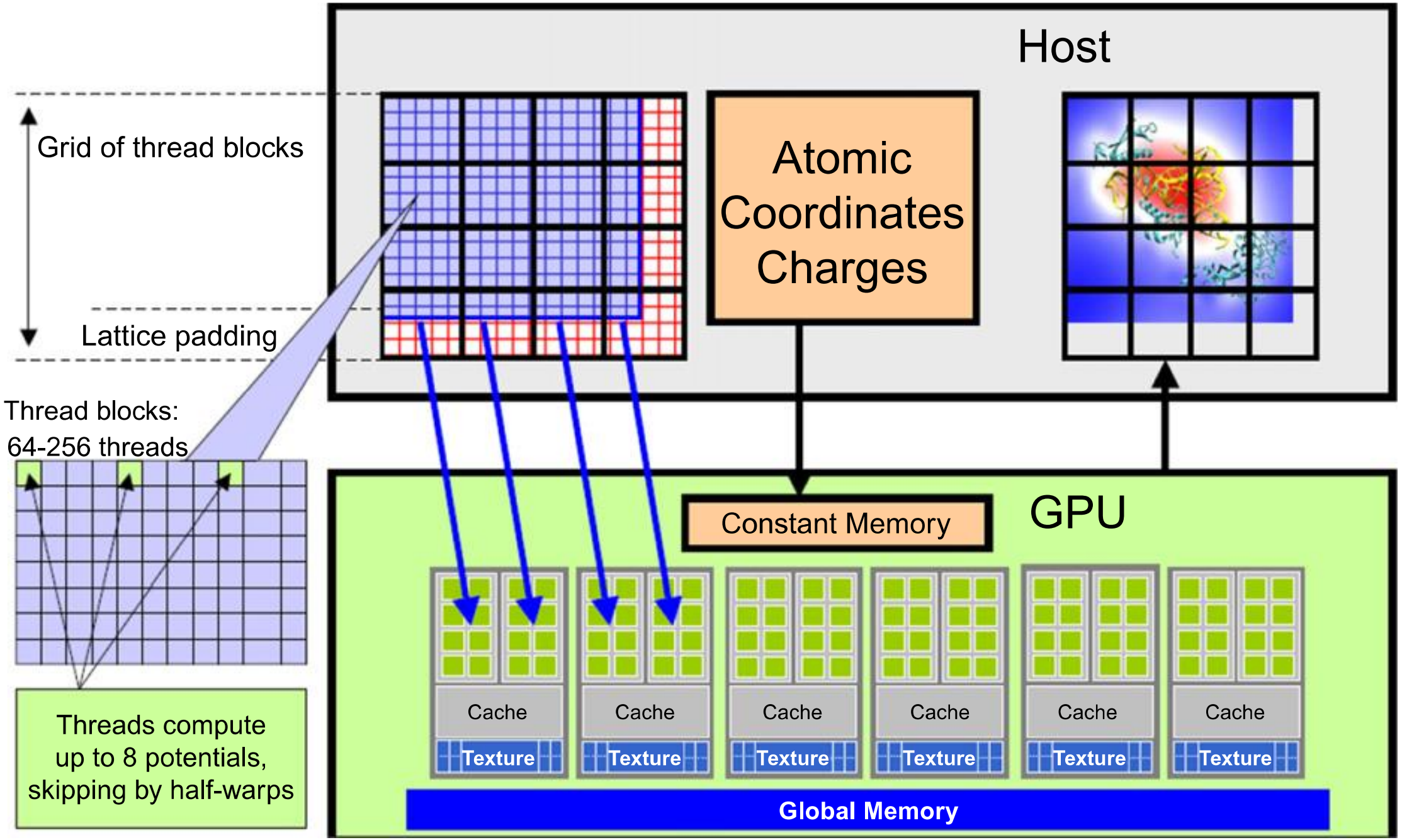
- 1) each thread computes the contribution of a particular atom to all grid points -> require much atomic memory operations
- 2) Each thread computes the contribution of all atoms for one grid point -> no need for atomic operations

Mapping iterations into a 2D thread grid

- 1) Make loop fission
  - Make an array to hold all y values, in a separate loop
  - Kernel launch overhead
- 2) Move y computation into the inner loop
  - More computations
  - More parallelism -> all iteration are executed in parallel (trade-off)

Features

- All atoms will be read by all threads
- Values of atoms are not changed -> can be placed in the constant memory


Host
Grid of thread blocks
Lattice padding
Atomic Coordinates Charges
Thread blocks: 64-256 threads
Threads compute up to 8 potentials, skipping by half-warps
Constant Memory
GPU
Cache
Texture
Global Memory

Granularity and chunking

- A thread can compute more than one grid point to improve data reuse
- The atoms are chunked and the kernel is invoked repetitively (to save global memory bandwidth)
- Hiding memory latency by reading a grid point and the start and writing it at the end
-

```
...
 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.

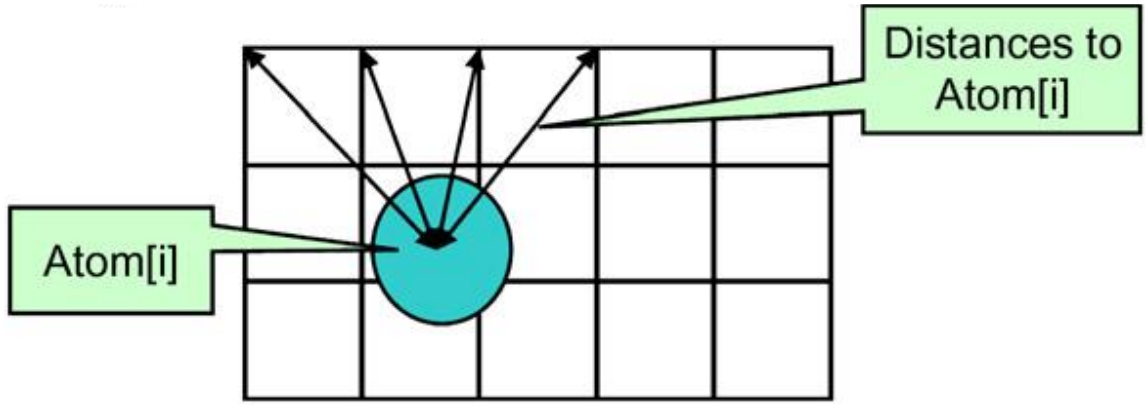
Only dependency on global memory read is at the end of the kernel…

Issues

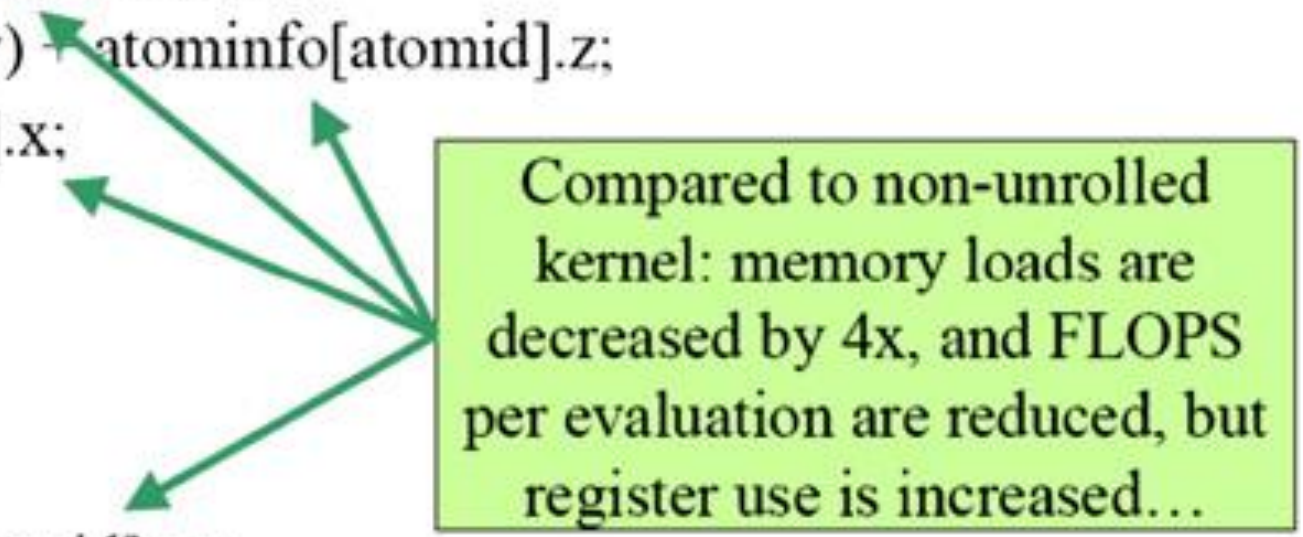
- Memory to computation ratio is 9 to 4, (better be 1 to 8), but cache can filter that
- However, the memory instruction can be better reused for increasing computation

Granularity

- Fuse many points together
- Also across a row, dy is the same

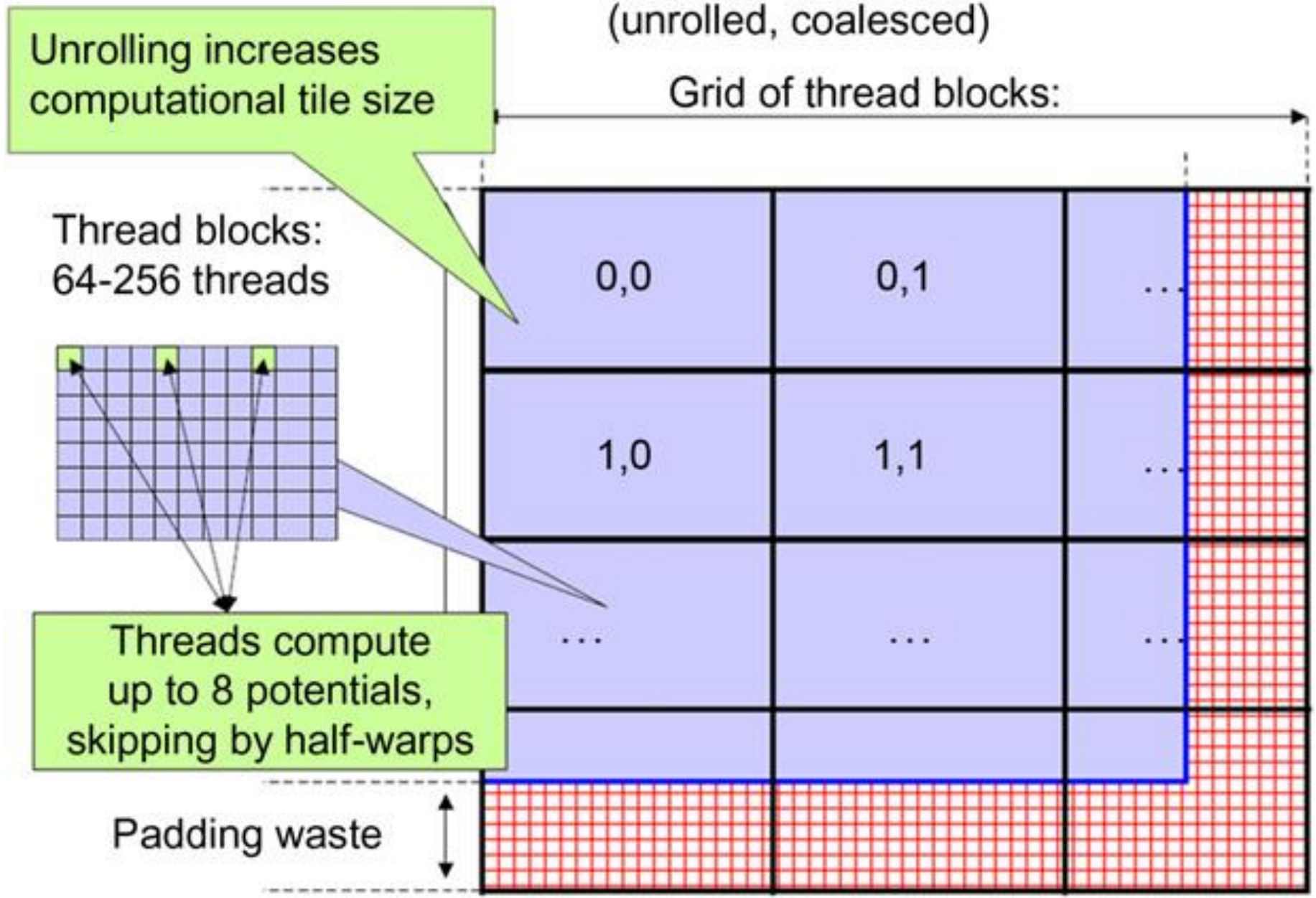


```
…for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float x = atominfo[atomid].x;
    float dx1 = coorx1 - x;
    float dx2 = coorx2 - x;
    float dx3 = coorx3 - x;
    float dx4 = coorx4 - x;
    float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



Memory Coalescing

- Access to memory is not coalesced for energygrid[] array


(unrolled, coalesced)
Unrolling increases computational tile size
Grid of thread blocks:
Thread blocks: 64-256 threads
0,0
0,1
1,0
1,1
Threads compute up to 8 potentials, skipping by half-warps
Padding waste

issues

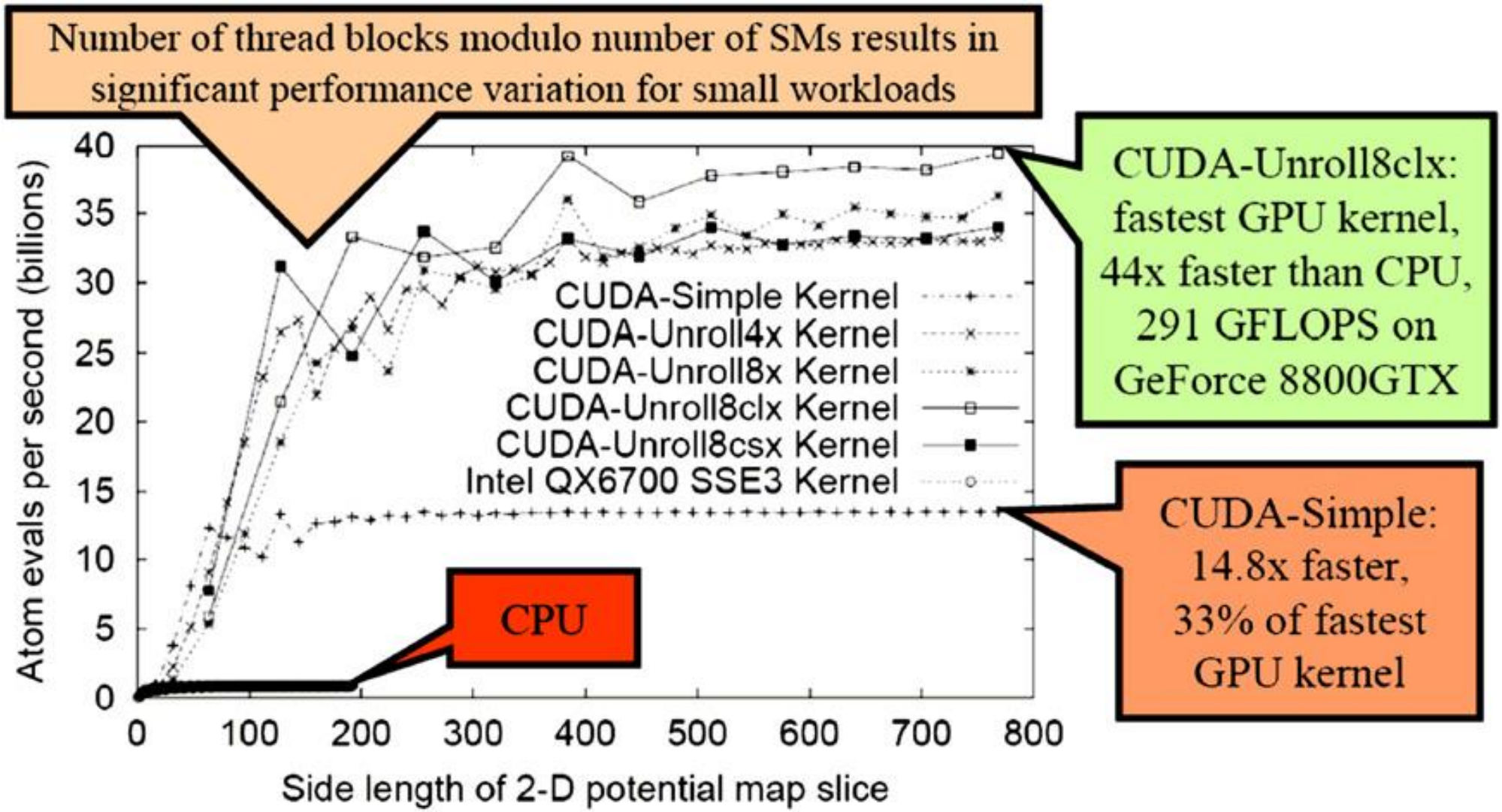
- Padding solves coalescing

```
…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



GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.