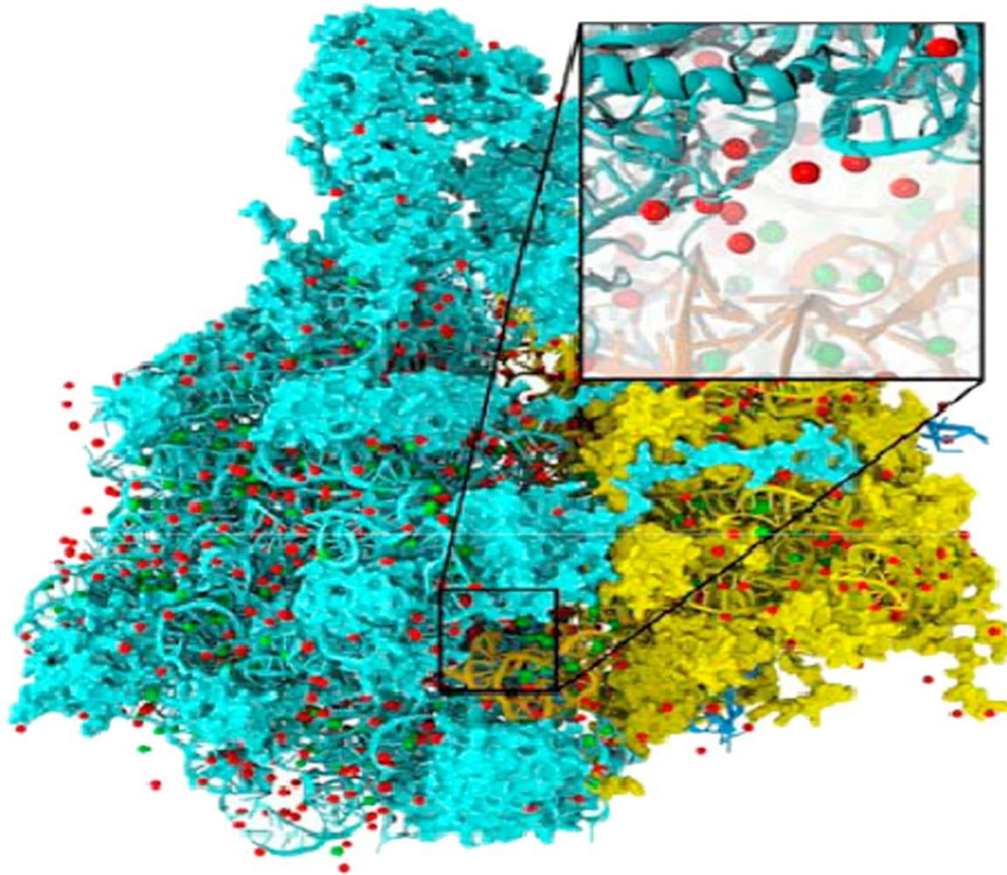
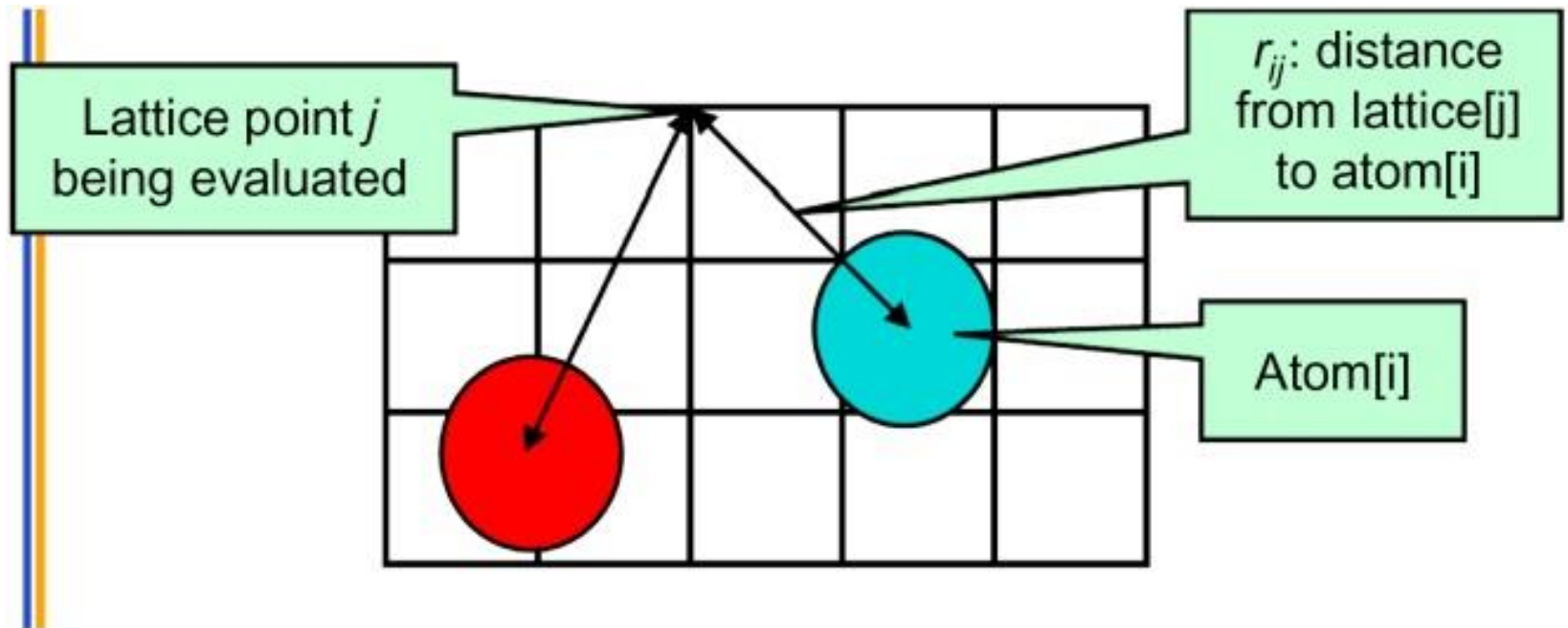


# Chapter-15

## Application case study— molecular visualization and analysis



**FIGURE 15.1:** Electrostatic potential map is used in building stable structures for molecular dynamics simulation.



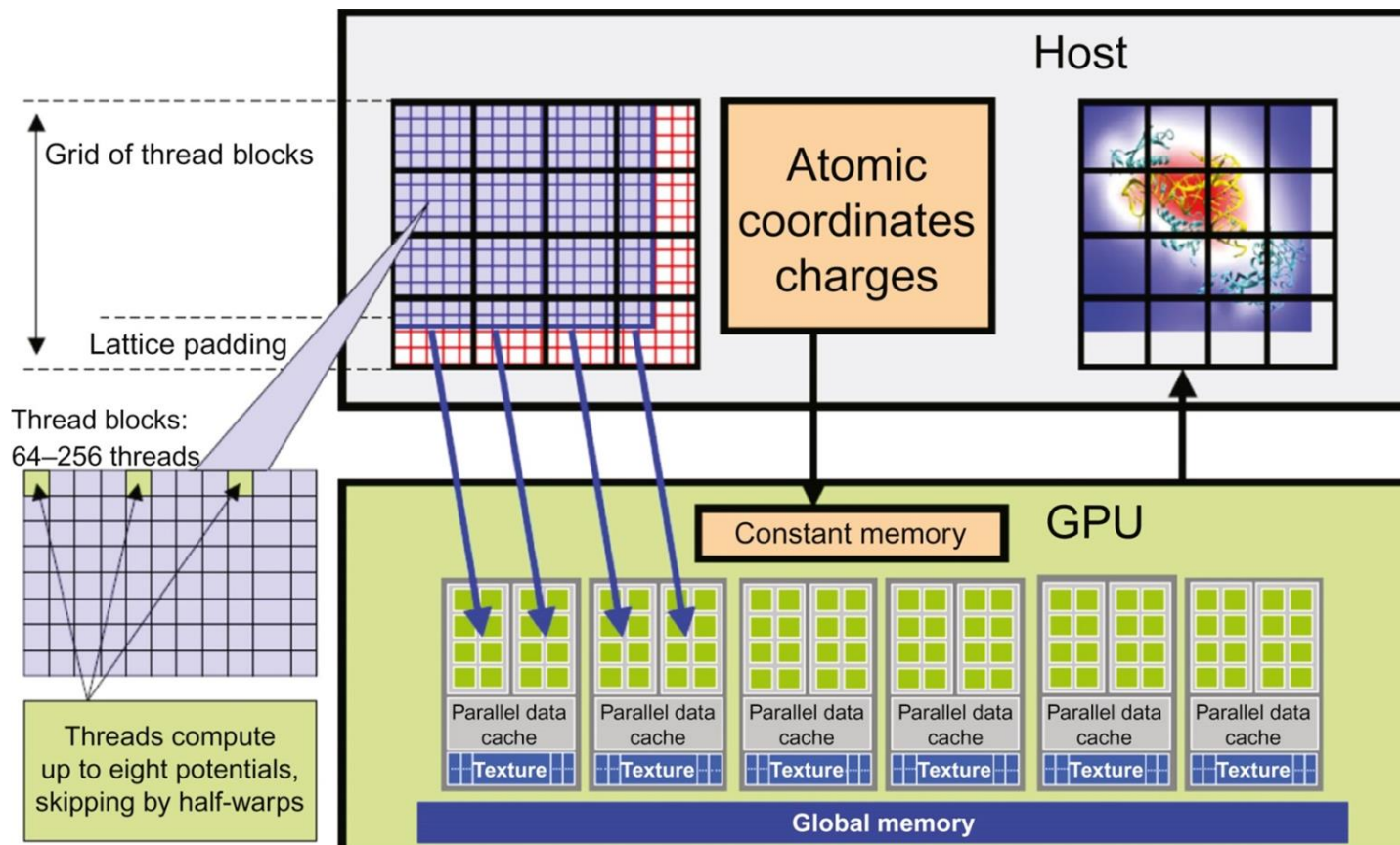
**FIGURE 15.2:** The contribution of atom[i] to the electrostatic potential at lattice point  $j$  (potential[j]) is atom[i] charge/ $r_{ij}$ . In the Direct Coulomb Summation method, the total potential at lattice point  $j$  is the sum of contributions from all atoms in the system.

```

void cenergy(float *energygrid, dim3 grid, float gridspace, float z, const float *atoms,
            int numatoms) {
    int i,j,n;
    int atomarrdim = numatoms * 4;
    for (j=0; j<grid.y; j++) {
        float y = gridspace * (float) j;
        for (i=0; i<grid.x; i++) {
            float x = gridspace * (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;
        }
    }
}

```

**FIGURE 15.3:** Base Coulomb potential calculation code for a 2D slice.



**FIGURE 15.4:** Overview of the DCS kernel design.

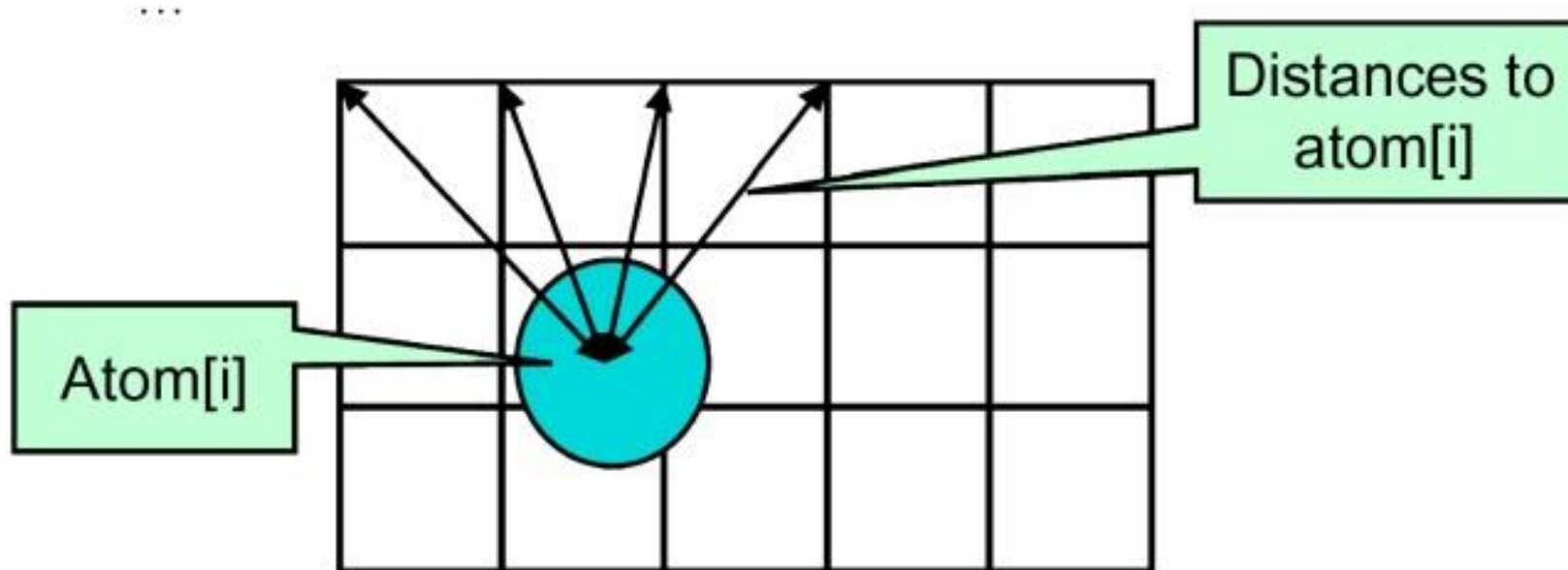
...

```
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...

**FIGURE 15.5:** DCS kernel version 1.



**FIGURE 15.6:** Reusing computation results among multiple grid points.



```

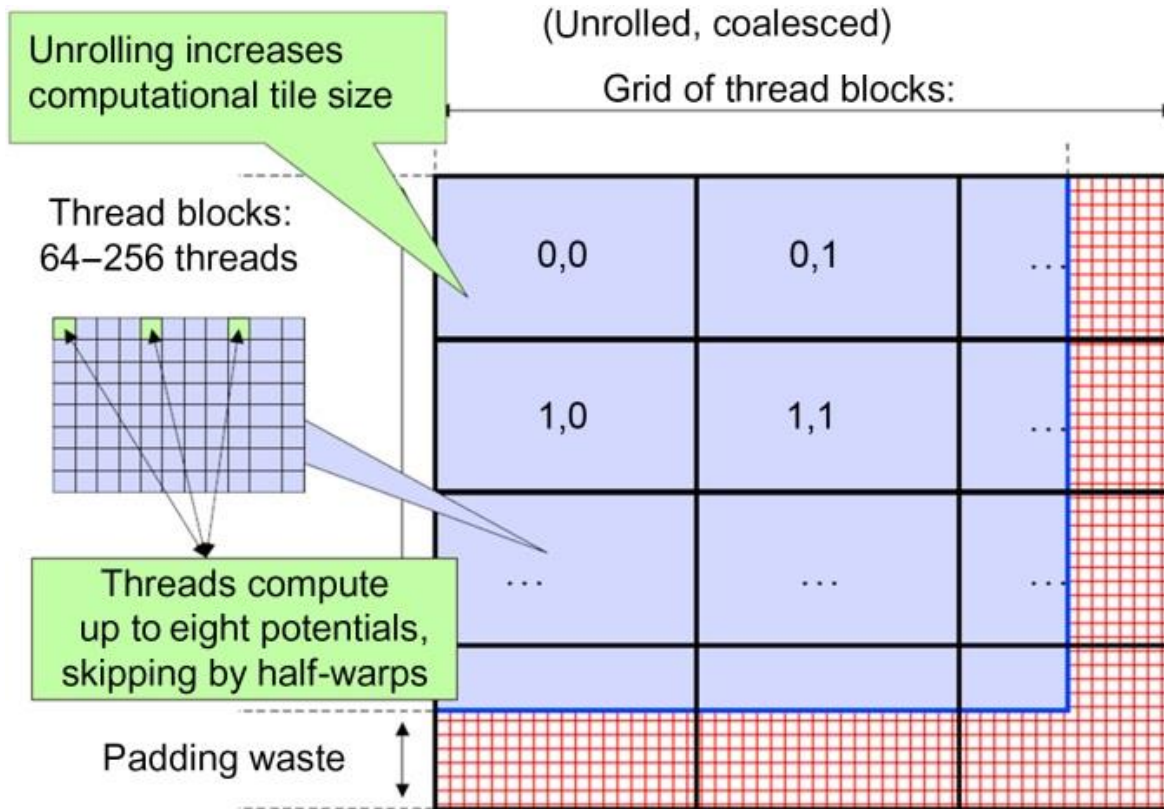
...for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float x = atominfo[atomid].x;
    float dx1 = coorx1 - x;
    float dx2 = coorx2 - x;
    float dx3 = coorx3 - x;
    float dx4 = coorx4 - x;
    float charge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

**FIGURE 15.7:** Version 2 of the DCS kernel.





**FIGURE 15.8:** Organizing threads and memory layout for coalesced writes.

```

...float coory = gridspacing * yindex;
float coorx = gridspacing * xindex;
float gridspacing_coalesce = gridspacing * BLOCKSIZE;
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;
[...]
```

Points spaced for memory coalescing

```

    float dx8 = dx7 + gridspacing_coalesce;
    energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
[...]
```

Reuse partial distance components  $dy^2 + dz^2$

```

    energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
}
energygrid[outaddr] += energyvalx1;
[...]
```

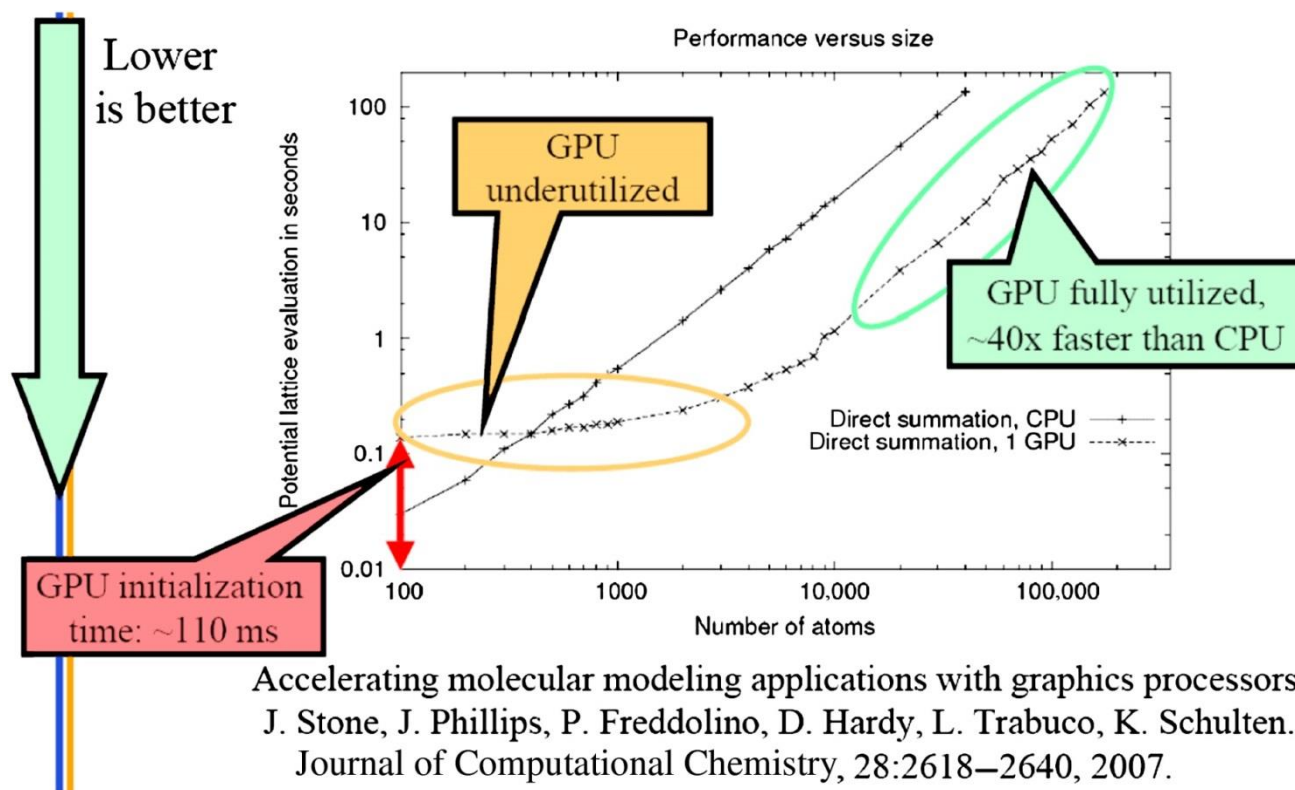
Global memory ops occur only at the end of the kernel, decreases register use

```

    energygrid[outaddr+7*BLOCKSIZE] += energyvalx7;

```

**FIGURE 15.9:** DCS kernel version 3.



**FIGURE 15.10:** Single-threads CPU versus CPU–GPU comparison.