GPU Teaching Kit

Accelerated Computing

Module 16 - Application Case Study – Electrostatic Potential Calculation

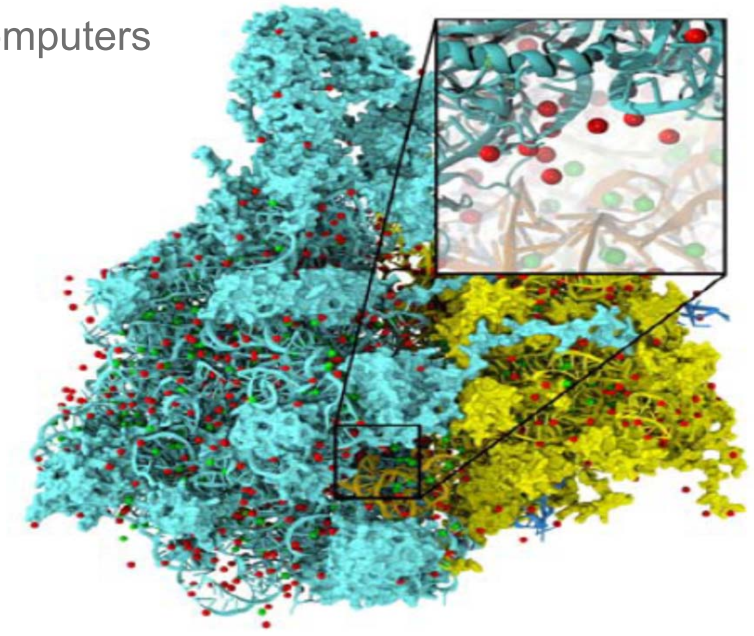
Lecture 16.1 - Electrostatic Potential Calculation - Part 1

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

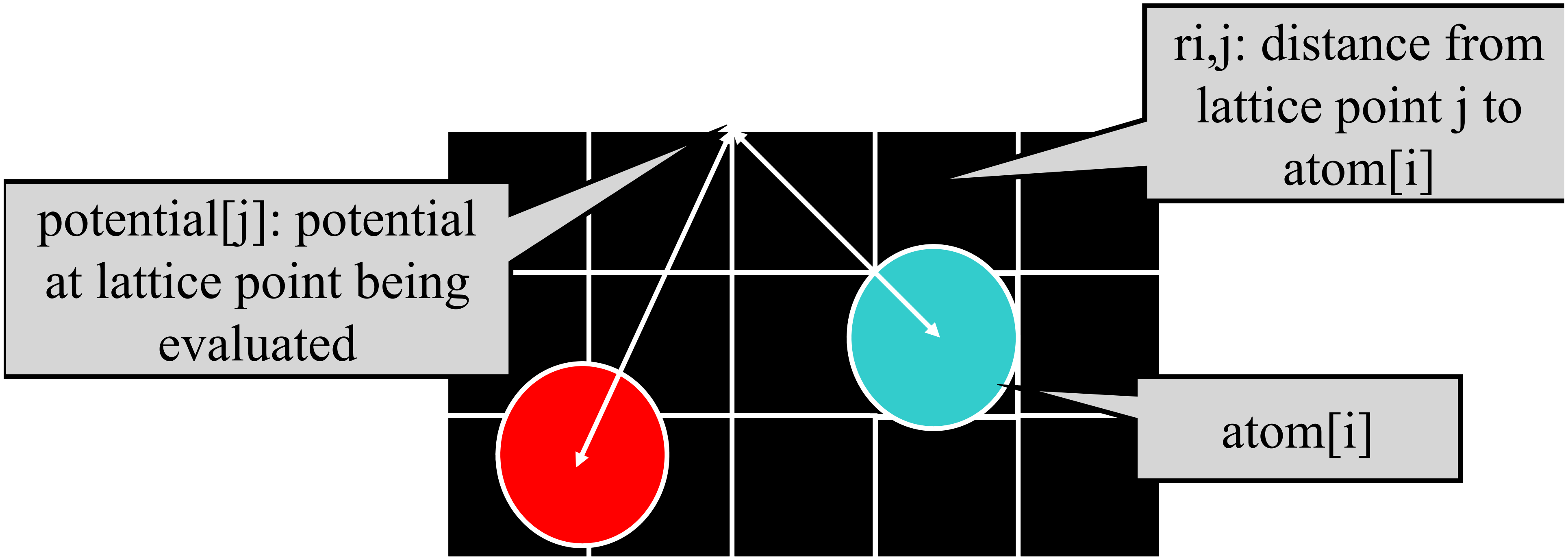
VMD

- Visual Molecular Dynamics
  - Visualizing, animating, and analyzing bio-molecular systems
  - More than 200,000 users as of 2012
  - Batch (movie making) vs. interactive mode
  - Run 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.


ri,j: distance from lattice point j to atom[i]
potential[j]: potential at lattice point being evaluated
atom[i]

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.

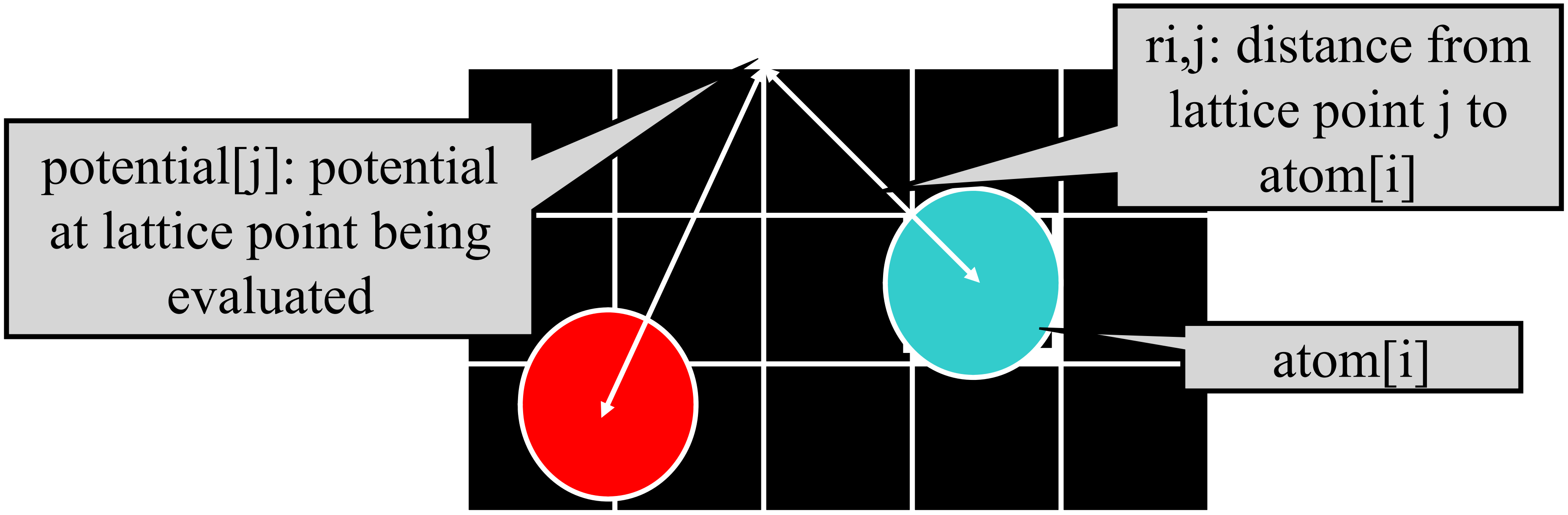
Overview of Direct Coulomb Summation (DCS) Algorithm

- 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

Direct Coulomb Summation (DCS) Algorithm Detail

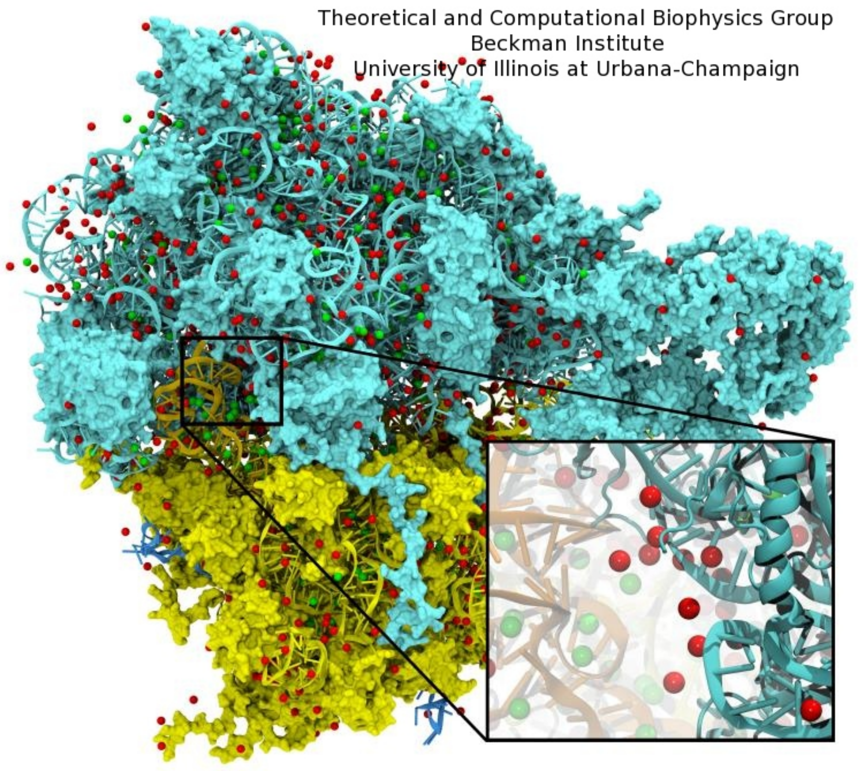
- At each lattice point, sum potential contributions for all atoms in the simulated structure:
  - potential[j] += charge[i] / (distance to atom[i])


potential[j]: potential at lattice point being evaluated
ri,j: distance from lattice point j to atom[i]
atom[i]

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


Theoretical and Computational Biophysics Group
Beckman Institute
University of Illinois at Urbana-Champaign

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 calculations on distances, coordinates, etc.

An Intuitive Sequential C Version

```
void cenergy (float *energygrid, dim3 grid, float gridspacing, float z,
const float * atoms, int numatoms) {
  int I, j, n;
  int k = z / gridspacing;
  int automarrdim = 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<automarrdim; n+=4) {
        float dx = x – atoms [n  ];
        float dy = y – atomas[n+1];
        float dz = z – atoms[n+2];
        energy += atomas[c+3]/sqrtf(dx * dx + dz * dz);
      }
      energygrid[grid.x * grid.y * k + grid.x * j + i= = energy;
    }
  }
}
```

The grid parameter gives the number of grid points in each dimension of the lattice.

A More Optimized Sequential C Version

Input oriented

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float
*atoms, int numatoms) {
  int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
  int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
for (int n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each atom
    float dz = z - atoms[n+2];  // all grid points in a slice have the same z  value
    float dz2 = dz*dz;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
      float y = gridspacing * (float) j;
      float dy = y - atoms[n+1];  // all grid points in a row have the same y value
      float dy2 = dy*dy;
      int grid_row_offset =  grid_slice_offset+ grid.x*j;
      for (int i=0; i<grid.x; i++) {
         float x = gridspacing * (float) i;
         float dx = x - atoms[n    ];
         energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2+ dz2);
      }
    }
  }
}
```

An More Optimized Sequential C Version

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

  int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
  for (int n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each
   atom
    float dz = z - atoms[n+2];  // all grid points in a slice have the same z  value
    float dz2 = dz*dz;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
      float y = gridspacing * (float) j;
      float dy = y - atoms[n+1];  // all grid points in a row have the same y value
      float dy2 = dy*dy;
      int grid_row_offset =  grid_slice_offset+ grid.x*j;
      for (int i=0; i<grid.x; i++) {
         float x = gridspacing * (float) i;
         float dx = x - atoms[n    ];
         energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2+ dz2);
      }
    }
  }
}
```

A More Optimized Sequential C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float
   *atoms, int numatoms) {
  int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
  int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
  for (int n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each
   atom
    float dz = z - atoms[n+2];  // all grid points in a slice have the same z  value
    float dz2 = dz*dz;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
      float y = gridspacing * (float) j;
      float dy = y - atoms[n+1];  // all grid points in a row have the same y value
      float dy2 = dy*dy;
      int grid_row_offset =  grid_slice_offset+ grid.x*j;
      for (int i=0; i<grid.x; i++) {
         float x = gridspacing * (float) i;
         float dx = x - atoms[n    ];
         energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2+ dz2);
      }
    }
  }
}
```

An Intuitive Sequential C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float
    *atoms, int numatoms) {
   int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
   int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
  for (int n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each
    atom
    float dz = z - atoms[n+2];  // all grid points in a slice have the same z  value
    float dz2 = dz*dz;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
      float y = gridspacing * (float) j;
      float dy = y - atoms[n+1];  // all grid points in a row have the same y value
      float dy2 = dy*dy;
      int grid_row_offset =  grid_slice_offset+ grid.x*j;
      for (int i=0; i<grid.x; i++) {
         float x = gridspacing * (float) i;
         float dx = x - atoms[n    ];
         energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2+ dz2);
      }
    }
  }
}
```

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

Straightforward CUDA Parallelization

- Use each thread to compute the contribution of an atom to all grid points in the current slice
  - Scatter parallelization
- Kernel code largely correspond to intuitive CPU version with outer loop stripped
  - Each thread corresponds to an outer loop iteration of CPU version
  - numatoms used in kernel launch configuration host code

A Very Slow DCS Scatter Kernel!

```
void  __global__ cenergy(float *energygrid, float *atoms, dim3 grid, float gridspacing,
float z) {
    int n = (blockIdx.x * blockDim .x + threadIdx.x) * 4;
    float dz = z - atoms[n+2];  // all grid points in a slice have the same z  value
    float dz2 = dz*dz;
    int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
      float y = gridspacing * (float) j;
      float dy = y - atoms[n+1];  // all grid points in a row have the same y value
      float dy2 = dy*dy;
      int grid_row_offset =  grid_slice_offset+ grid.x*j;
      for (int i=0; i<grid.x; i++) {
         float x = gridspacing * (float) i;
         float dx = x - atoms[n    ];
         energygrid[grid_row_offset + i]  += charge / sqrtf(dx*dx + dy2+ dz2));
      }
    }
  }
}
```

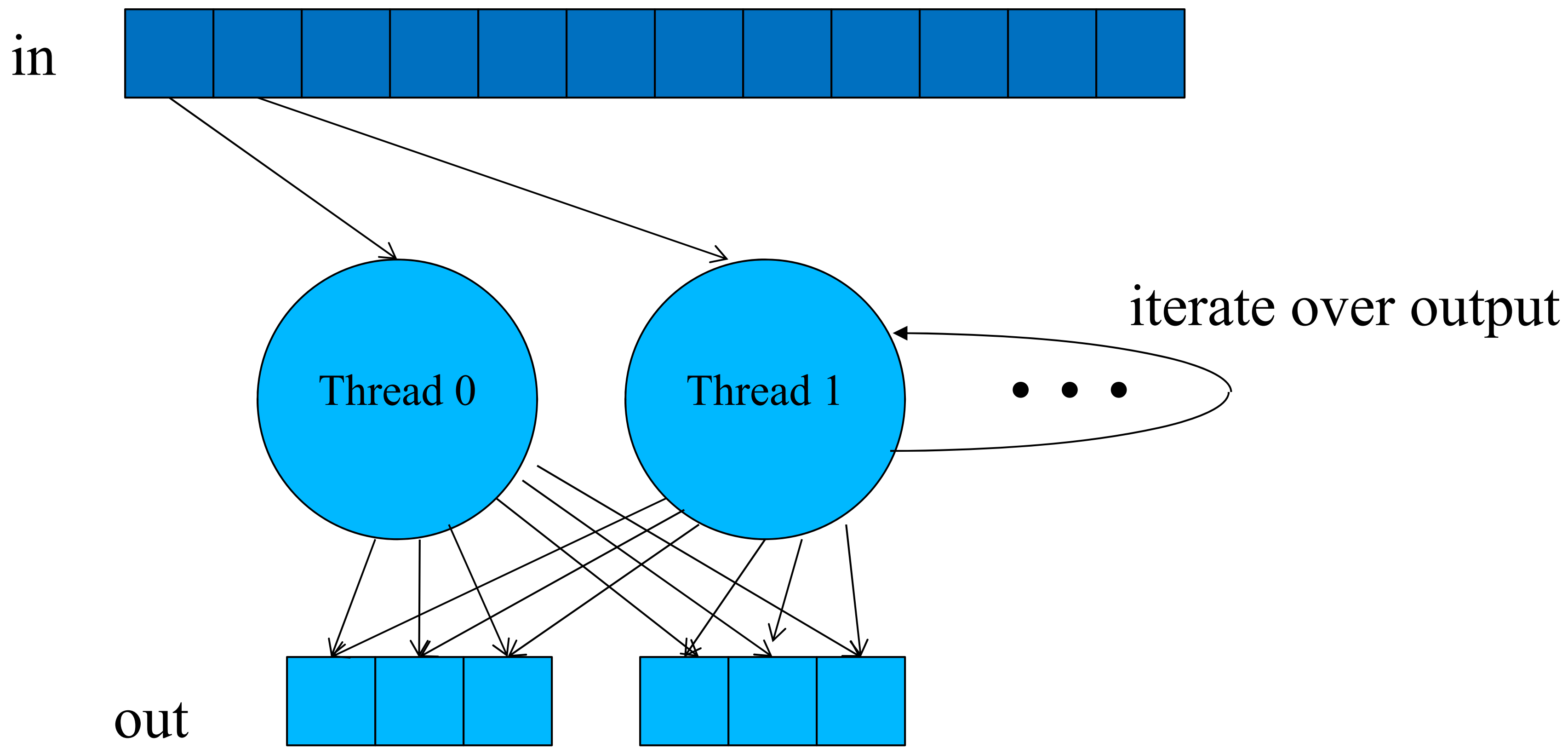
Needs to be calculated redundantly by every thread

A Very Slow DCS Scatter Kernel!

```
void  __global__ cenergy(float *energygrid, float *atoms, dim3
grid, float gridspacing, float z) {
    int n = (blockIdx.x * blockDim .x + threadIdx.x) *4;
    float dz = z - atoms[n+2];  // all grid points in a slice have
the same z  value
    float dz2 = dz*dz;
    int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
      float y = gridspacing * (float) j;
      float dy = y - atoms[n+1];  // all grid points in a row have
the same y value
      float dy2 = dy*dy;
      int grid_row_offset =  grid_slice_offset+ grid.x*j;
      for (int i=0; i<grid.x; i++) {
         float x = gridspacing * (float) i;
         float dx = x - atoms[n    ];
         energygrid[grid_row_offset + i]  += charge / sqrtf(dx*dx
+ dy2+ dz2));
      }
    }
  }
```

Needs to be done as an atomic operation

Scatter Parallelization



Why is scatter parallelization often used rather than gather?

- In practice, each in element does not have significant effect on all out elements
- Output tends to be much more regular than input
  - Input usually comes as sparse data structure, where coordinates are part of the data
  - One needs to look at the input data to see if an input is relevant to an output value
  - Output is usually a regular, grid
  - Given an input value, one can easily find output via index calculation

Challenges in Gather Parallelization

- Regularize input elements so that it is easier to find all in elements that affects an out element
  - Input Binning (ECE598HK)

- Can be even more challenging if data is non-uniformly distributed
  - Cut-off Binning for Non-Uniform Data (ECE598HK)

- For this lecture, we assume that all in elements affect all out elements

Pros and Cons of the Scatter Kernel

- Pros
  - Follows closely the simple CPU version
  - Good for software engineering and code maintenance
  - Preserves computation efficiency (coordinates, distances, offsets) of sequential code
- Cons
  - The atomic add serializes the execution, very slow!
  - Not even worth trying this.

GPU Teaching Kit

Accelerated Computing