

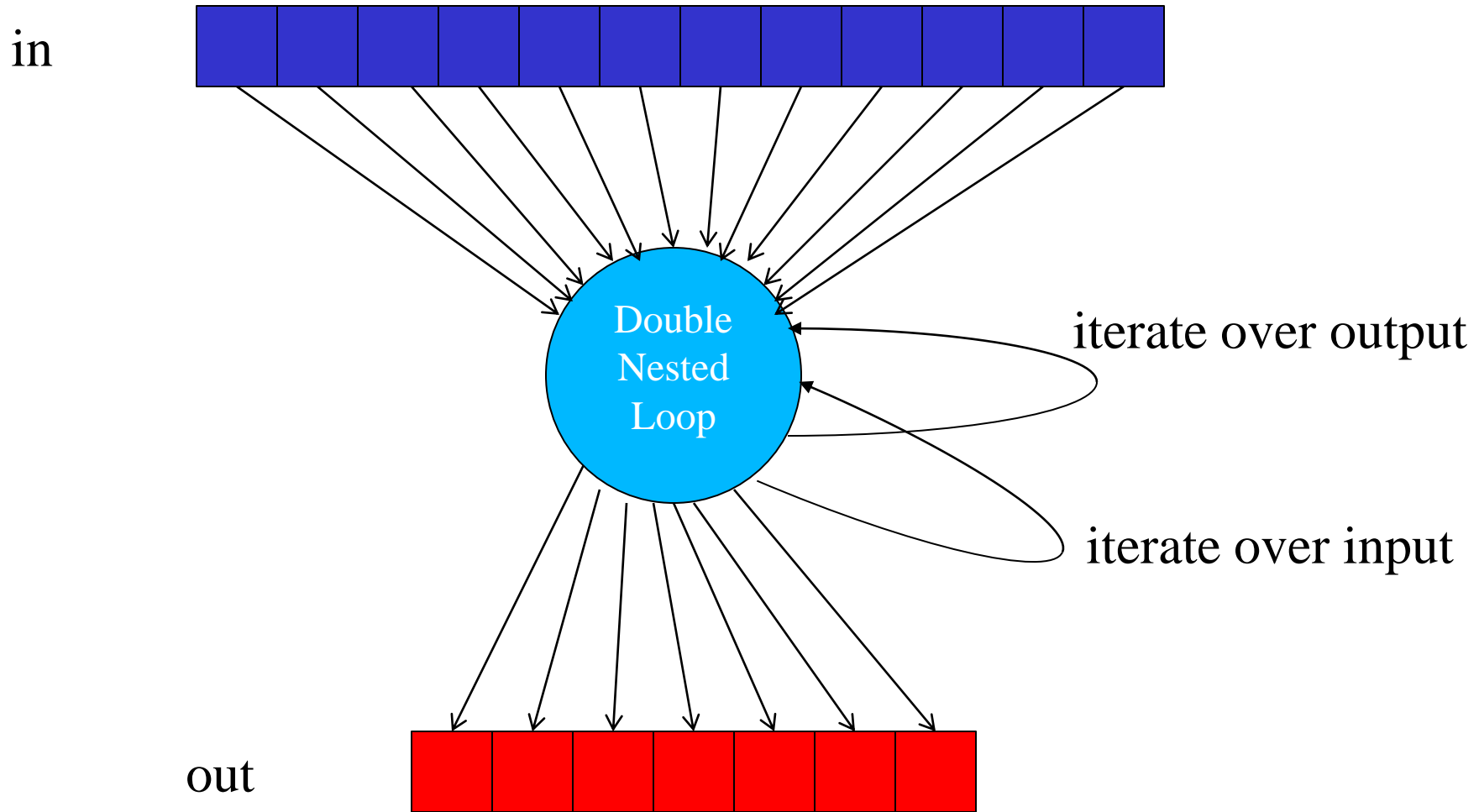


ICCS School

Advanced GPU Programming for Science

Lecture 4:
Scatter to Gather Transformation

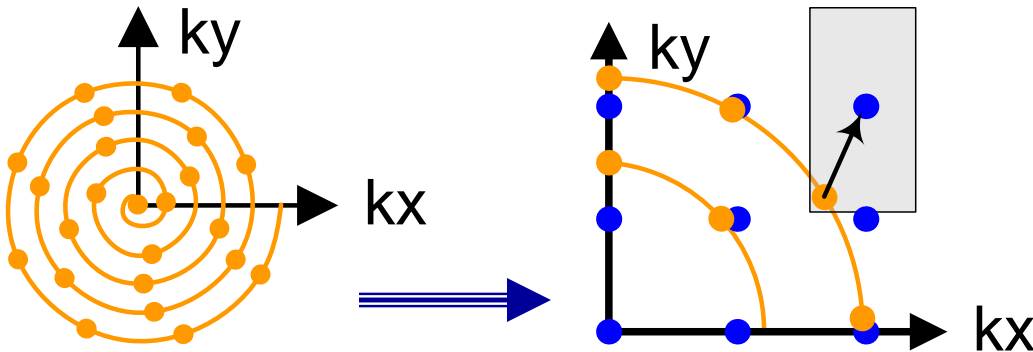
A Common Sequential Computation Pattern



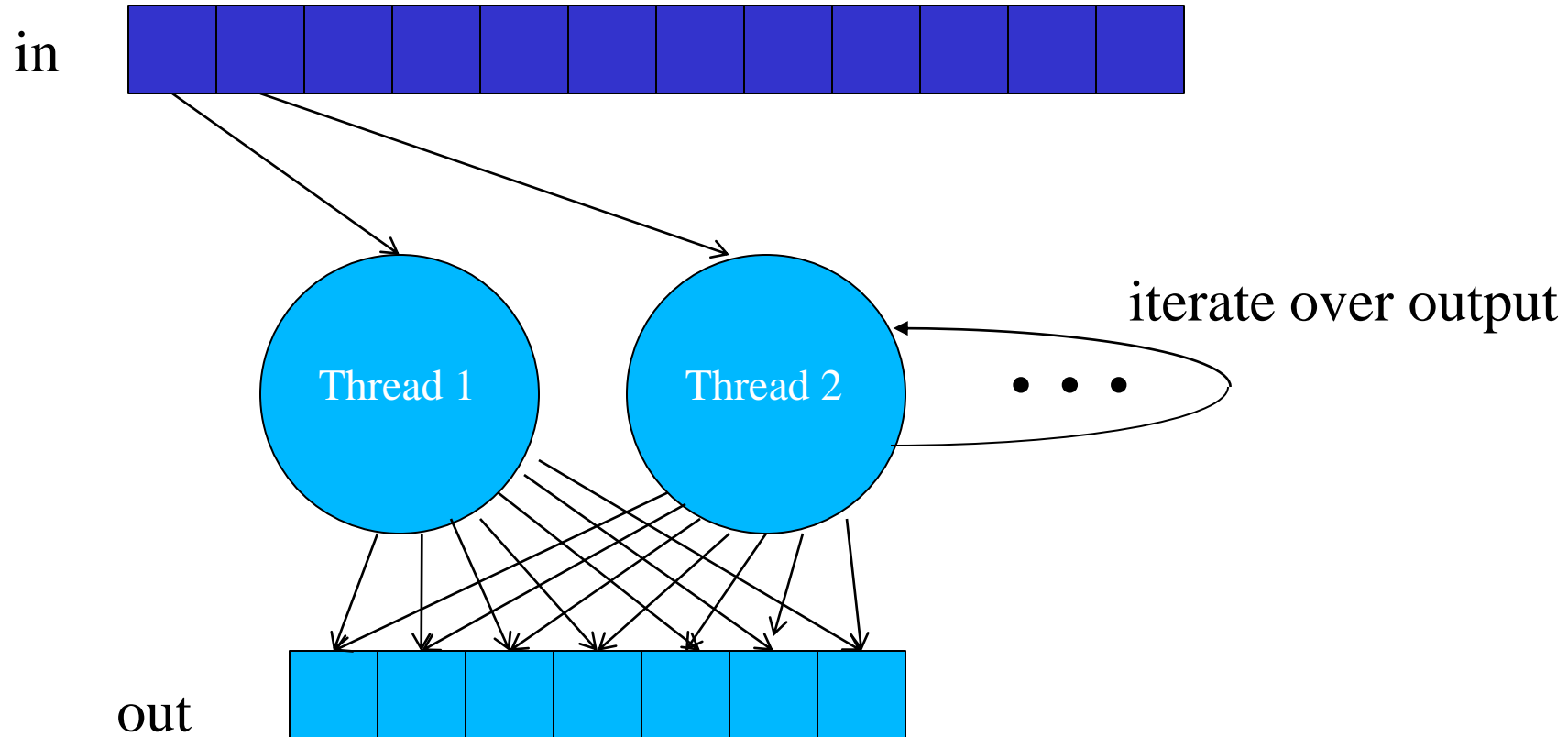
A Simple Code Example

```
for (m = 0; m < M; m++) {  
    for (n = 0; n < N; n++) {  
        out[n] += f(in[m], m, n);  
    }  
}
```

- Input data in
 - M = # scan points
- Output data out
 - N = # regularized scan points
- Complexity is $O(MN)$

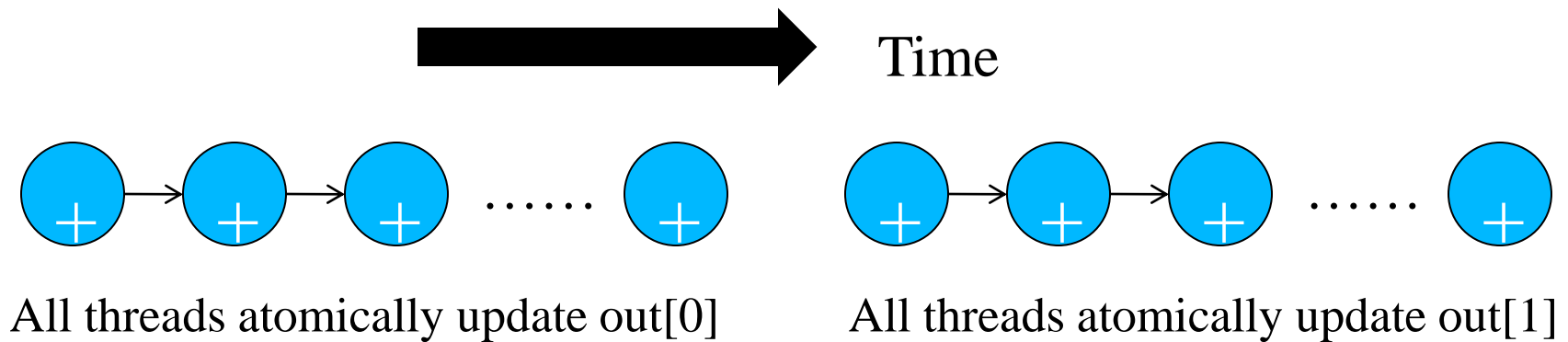


Scatter Parallelization



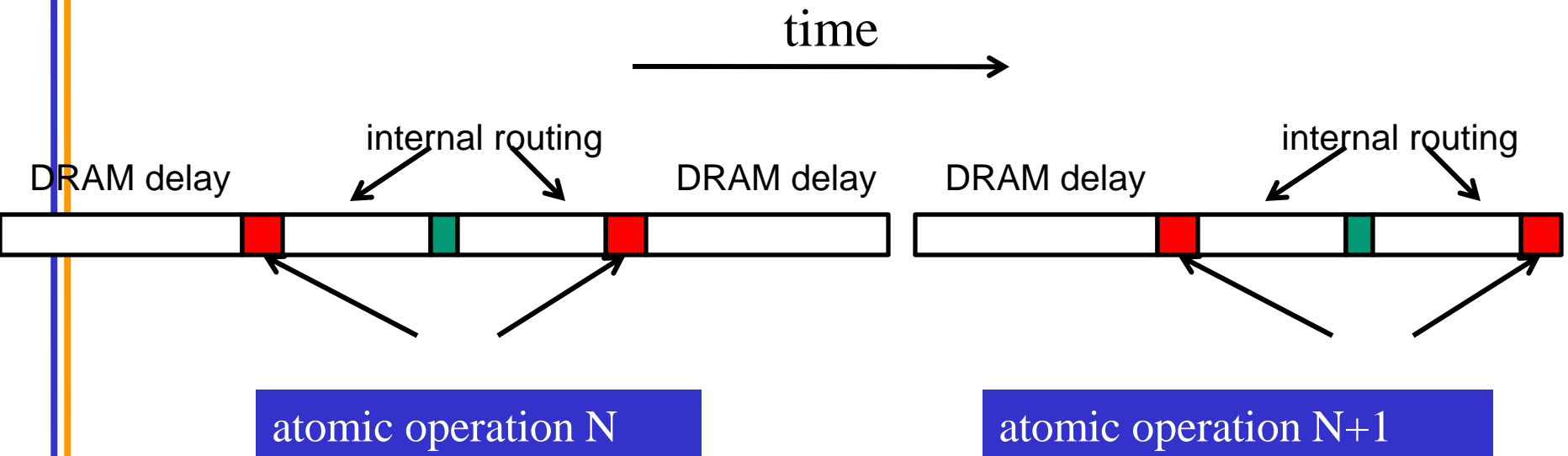
Scatter can be very slow.

- All threads have conflicting updates to the same out elements
 - Serialized with atomic operations
 - Very costly (slow) for large number of threads



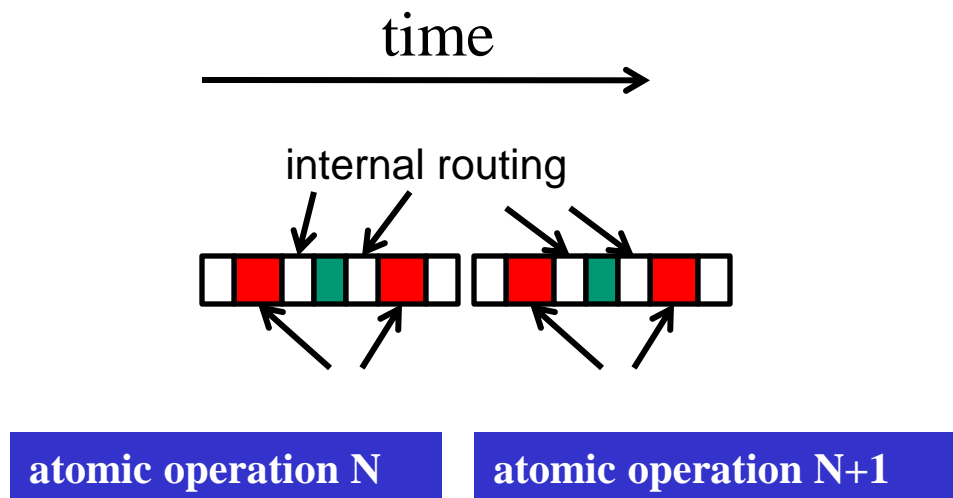
Atomic Operations on DRAM

- Each Load-Modify-Store has two full memory access delays
 - All atomic operations on the same variable (RAM location) are serialized



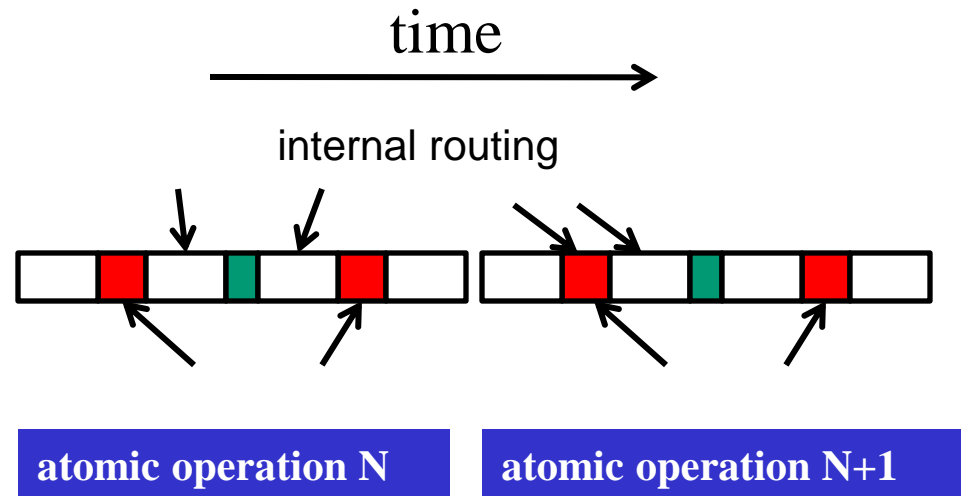
Hardware Improvements

- Atomic operations on Shared Memory
 - Very short latency, but still serialized
 - Private to each thread block
 - Need algorithm work by programmers (more later)

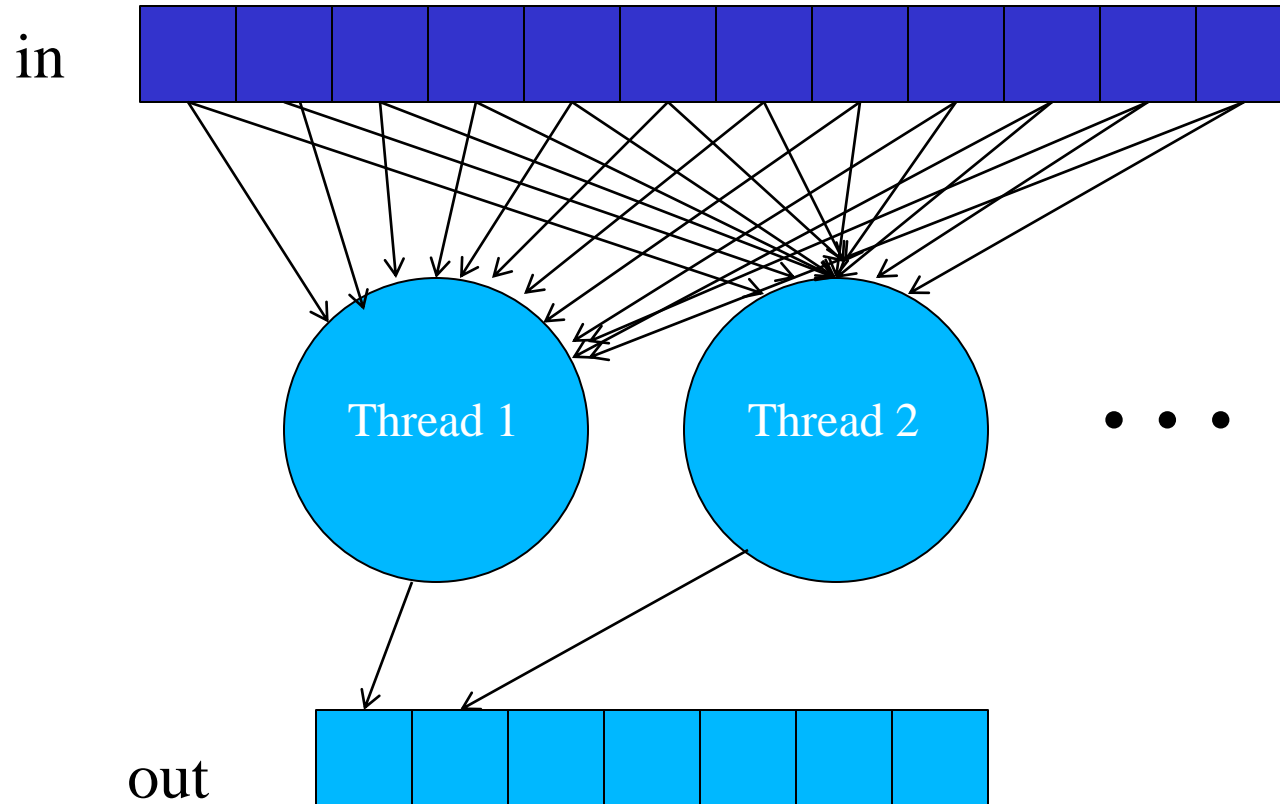


Hardware Improvements (cont.)

- Atomic operations on Fermi L2 cache
 - medium latency, but still serialized
 - Global to all blocks
 - “Free improvement” on Global Memory atomics

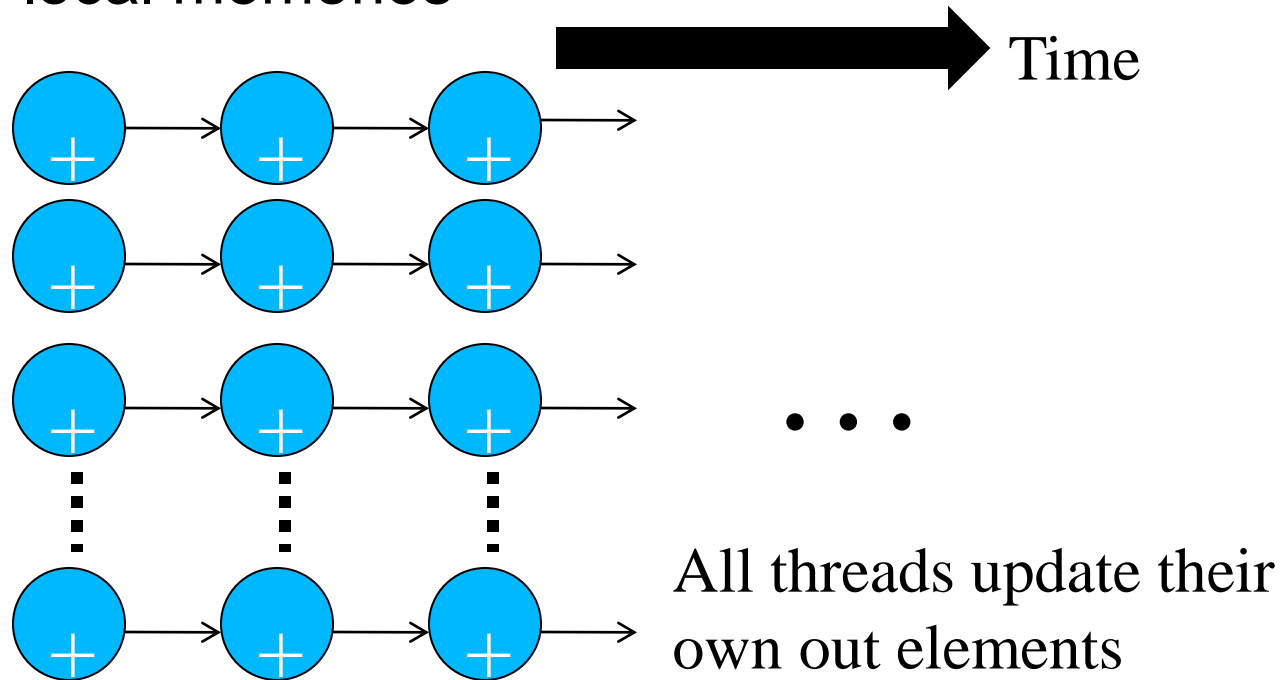


Gather Parallelization



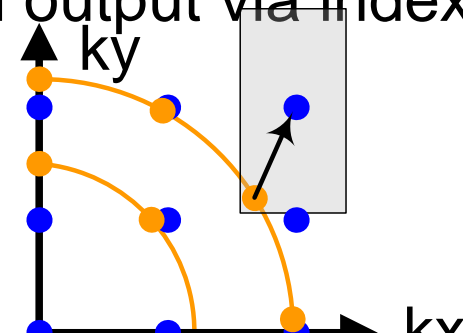
Gather can be very fast.

- All threads can read the same in elements
 - No serialization
 - Can even be efficiently consolidated through caches or local memories



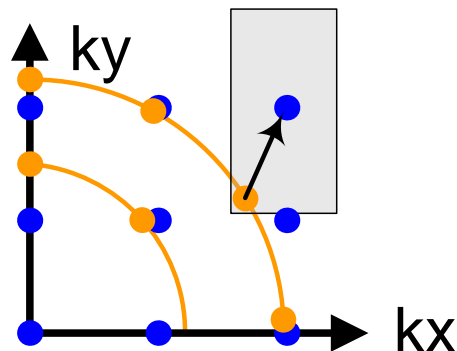
Why is scatter parallelization often used rather than gather?

- In practice, each input element does not affect all output 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 find output via index calculation



Why is scatter parallelization often used rather than gather?

- It is easy to calculate all out elements affected by an in element
 - Harder to calculate all in elements that affect an out
 - Easy thread kernel code if written in scatter

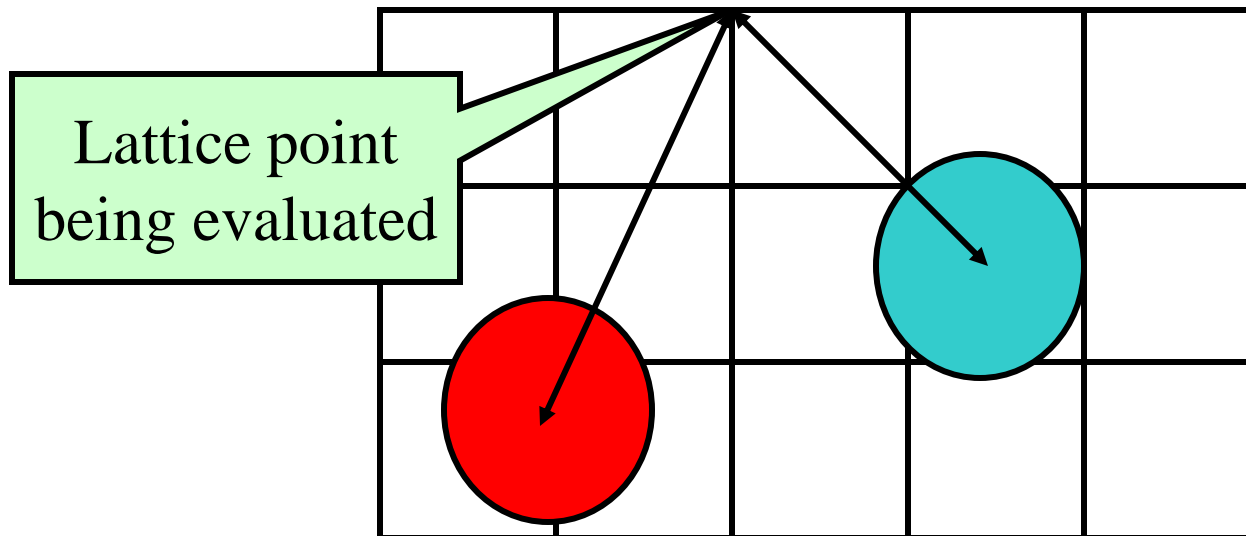


Challenges in Gather Parallelization

- Regularize input elements so that it is easier to find all in elements that affects an out element
 - Input Binning Lecture
- Can be even more challenging if data is highly non-uniform
 - Cut-off Binning for Non-Uniform Data Lecture
- For this lecture, we assume that all in elements affect all out elements

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.



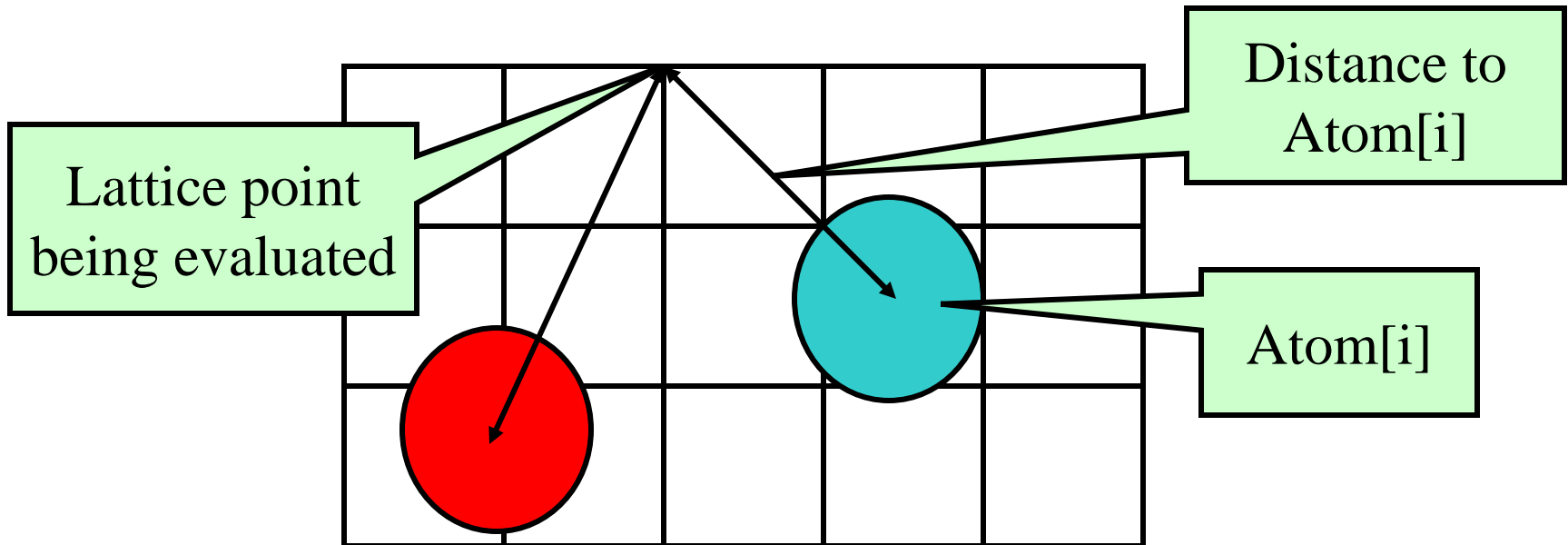
Overview of Direct Coulomb Summation (DCS) Algorithm

- 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:
$$\text{potential} += \text{charge}[i] / (\text{distance to atom}[i])$$
- Approximation-based methods such as cut-off summation can achieve much higher performance at the cost of some numerical accuracy and flexibility
 - Will cover these later

Direct Coulomb Summation (DCS) Algorithm Detail

- At each lattice point, sum potential contributions for all atoms in the simulated structure:

$$\text{potential} += \text{charge}[i] / (\text{distance to atom}[i])$$



Electrostatic Potential Map Calculation Function Overview

- Each call calculates an x-y slice of the energy map
 - *energygrid* – pointer to the entire potential map
 - *grid* – the x, y, z dimensions of the potential map
 - *gridspacing* – modeled physical dist between grid points
 - *atoms* – array of x, y, z coordinates and charge of atoms
 - *numatoms* – number of atoms in atoms array

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

An Intuitive Sequential C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspace, 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;
        int grid_slice_offset = (grid.x*grid.y*z) / gridspace;
        float charge = atoms[n+3];
        for (int j=0; j<grid.y; j++) {
            float y = gridspace * (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 = gridspace * (float) i;
                float dx = x - atoms[n ];
                energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2 + dz2);
            }
        }
    }
}
```

Input oriented

An Intuitive Sequential C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspace, 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;
        int grid_slice_offset = (grid.x*grid.y*z) / gridspace;
        float charge = atoms[n+3];
        for (int j=0; j<grid.y; j++) {
            float y = gridspace * (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 = gridspace * (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 gridspace, 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;
        int grid_slice_offset = (grid.x*grid.y*z) / gridspace;
        float charge = atoms[n+3];
        for (int j=0; j<grid.y; j++) {
            float y = gridspace * (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 = gridspace * (float) i;
                float dx = x - atoms[n ];
                energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2 + dz2);
            }
        }
    }
}
```

An Intuitive Sequential C Version

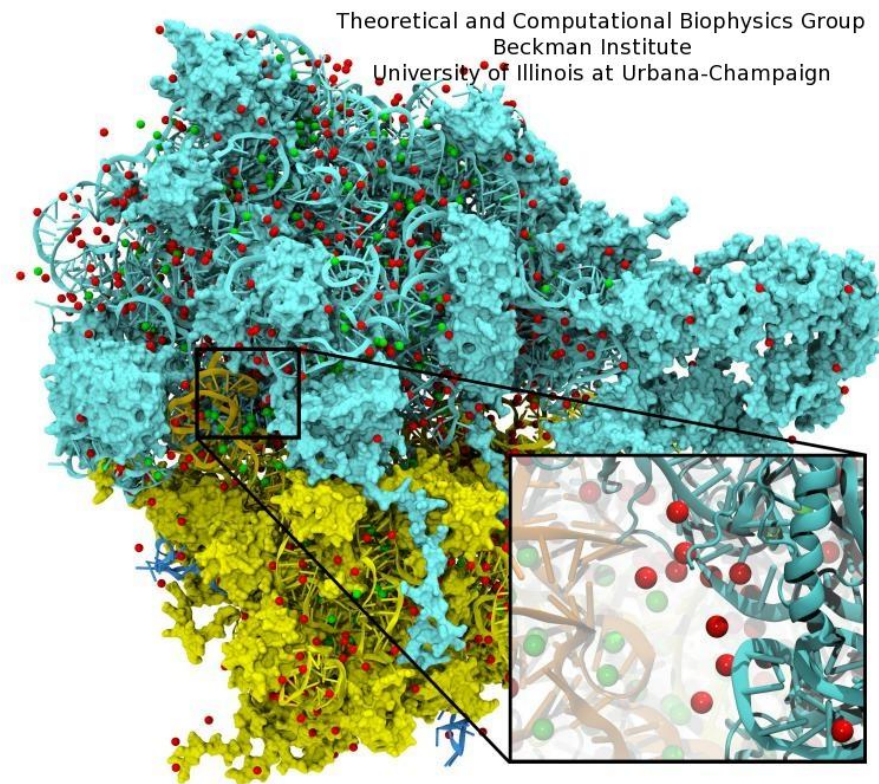
```
void cenergy(float *energygrid, dim3 grid, float gridspace, float z, const float *atoms,
            int numatoms) {
    int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
    for (int n=0; n<atomarrdim; n+=4) { // calculate potential contribution of each atom
        float dz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

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 (energygrid) is very 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.

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



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 gridspaceing, 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) / gridspaceing;
    float charge = atoms[n+3];
```

```
    for (int j=0; j<grid.y; j++) {
        float y = gridspaceing * (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 = gridspaceing * (float) i;
            float dx = x - atoms[n ];
            energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2+ dz2));
        }
    }
```

```
    }
}
```

A Very Slow DCS Scatter Kernel!

```
void __global__ cenergy(float *energygrid, float *atoms, dim3 grid, float gridspaceing,
    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) / gridspaceing;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
        float y = gridspaceing * (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 = gridspaceing * (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

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 yourself.

A Slower Sequential C Version

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

```
    int atomarrdim = numatoms * 4;
```

```
    int k = z / gridspace;
```

```
    for (int j=0; j<grid.y; j++) {
```

```
        float y = gridspace * (float) j;
```

```
        for (int i=0; i<grid.x; i++) {
```

```
            float x = gridspace * (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;
```

```
        }
```

```
    }
```

```
}
```

Output oriented.

A Slower Sequential C Version

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

    int atomarrdim = numatoms * 4;
    int k = z / gridspace;
    for (int j=0; j<grid.y; j++) {
        float y = gridspace * (float) j;
        for (int i=0; i<grid.x; i++) {
            float x = gridspace * (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;
        }
    }
}
```

More redundant work.

Pros and Cons of the Slower Sequential Code

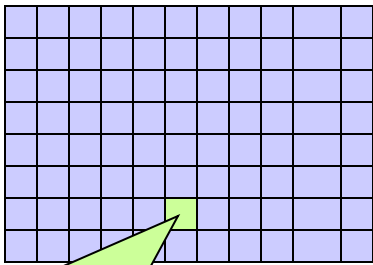
- Pros
 - Fewer access to the energygrid array
 - Simpler code structure
- Cons
 - Many more calculations on the coordinates
 - More access to the atom array
 - Overall, much slower sequential execution due to the sheer number of calculations performed

DCS CUDA Block/Grid Decomposition

(no register tiling)

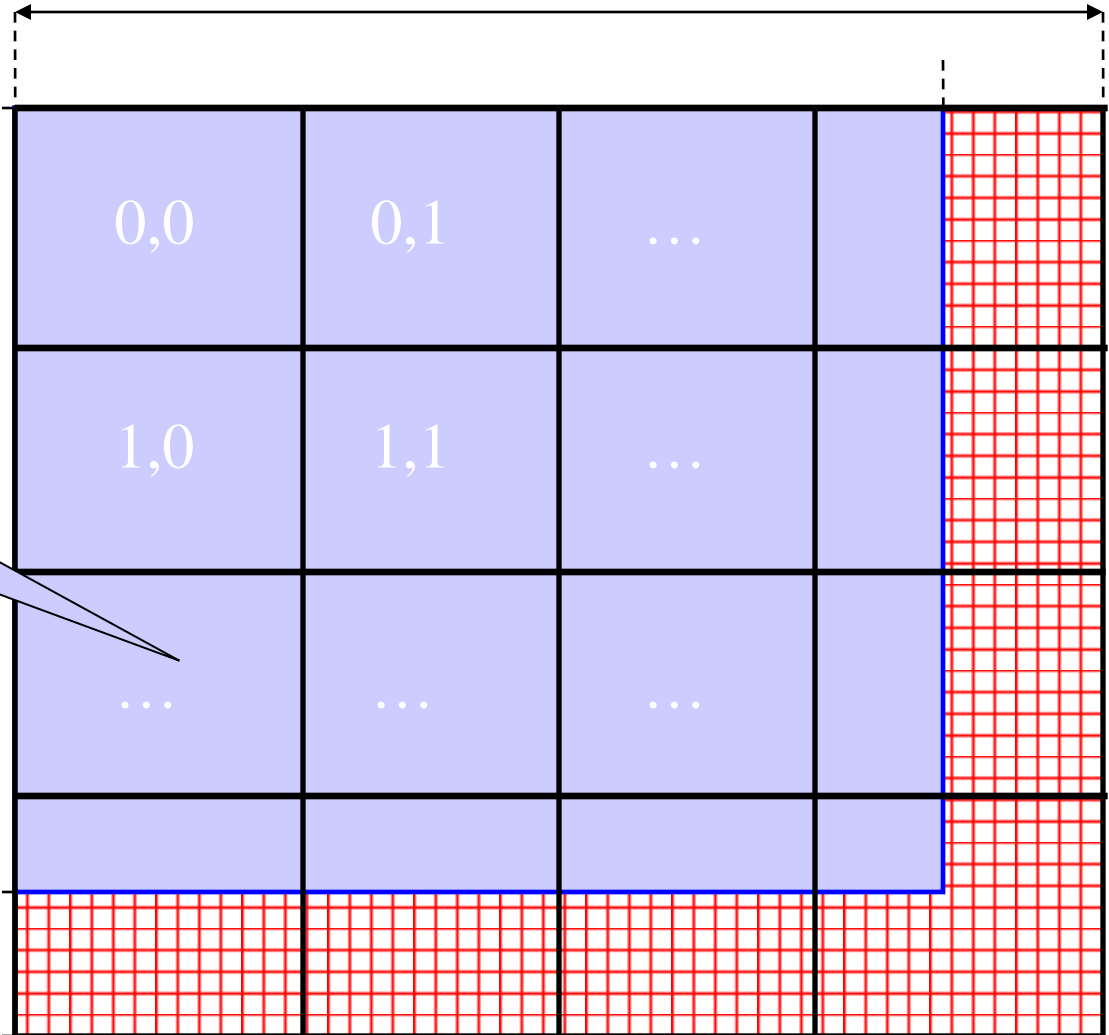
Grid of thread blocks:

Thread blocks:
64-256 threads



Threads compute
1 potential each

Padding waste



A Fast DCS CUDA Gather Kernel

```
void __global__ cenergy(float *energygrid, dim3 grid, float gridspace, 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 / gridspace;
```

```
    float y = gridspace * (float) j;
```

```
    float x = gridspace * (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;
```

```
}
```

One thread per grid point

A Fast DCS CUDA Gather Kernel

```
void __global__ cenergy(float *energygrid, dim3 grid, float gridspace, 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 / gridspace;
```

```
    float y = gridspace * (float) j;
```

```
    float x = gridspace * (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;
```

```
}
```

All threads access all atoms.

Consolidated writes to grid points

Additional Comments

- Further optimizations
 - dz^*dz can be pre-calculated and sent in place of z
- Gather kernel is much faster than a scatter kernel
 - No serialization due to atomic operations
- Compute efficient sequential algorithm does not translate into the fast parallel algorithm
 - Gather vs. scatter is a big factor
 - But we will come back to this point later!

Even More Comments

- In modern CPUs, cache effectiveness is often more important than compute efficiency
- The input oriented (scatter) sequential code actually has very bad cache performance
 - energygrid[] is a very large array, typically 20X or more larger than atom[]
 - The input oriented sequential code sweeps through the large data structure for each atom, trashing cache.
- The fastest sequential code is actually an optimized output oriented code

Outline of A Fast Sequential Code

for all z {

 for all atoms {precompute dz^2 }

 for all y {

 for all atoms {precompute $dy^2 (+ dz^2)$ }

 for all x {

 for all atoms {

 compute contribution to current x,y,z point
 using precomputed dy^2 and dz^2

 }

 } } }

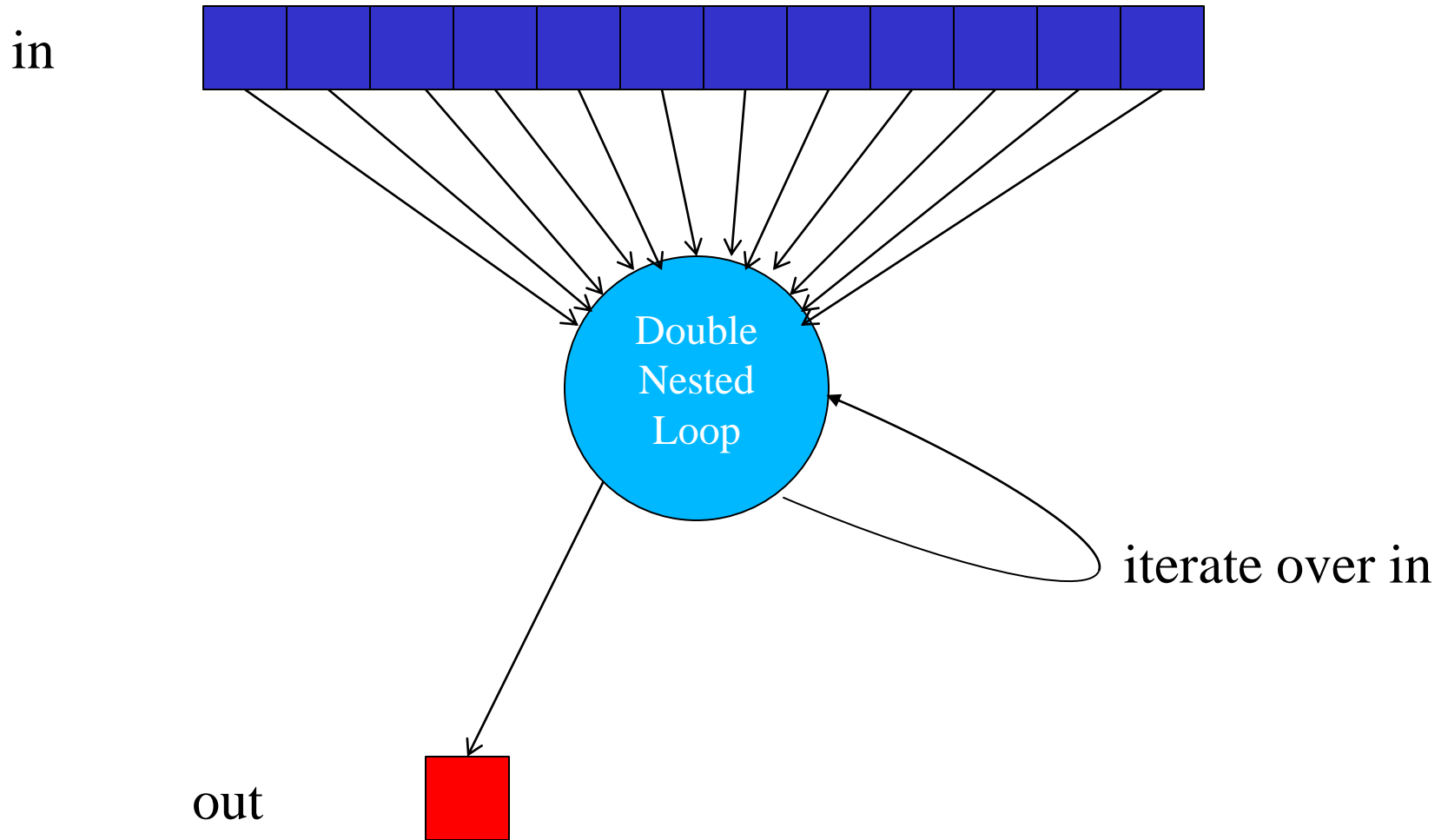
More Thoughts on Fast Sequential Code

- Need temporary arrays for pre-calculated dz^2 and $dy^2 + dz^2$ values
- So, why does this code has better cache behavior on CPUs?

A decorative element consisting of two vertical lines, one blue and one orange, running down the left side of the slide.

QUESTIONS?

Reduction – A Degenerate Case



There is no output parallelism!

- There is only one output
- But scatter style code is not acceptable
 - Each threads reads one input and accumulate into one reduction variable with atomic operation
 - All input threads write to ONE output location
- Tree reduction makes more sense

Solution – Create Multiple Outputs

