# ECE 508
# Manycore Parallel Algorithms

## Lecture 2:
## Scatter-to-Gather Transformation

# Objective

- to understand the performance implications of parallelization over inputs (‘scatter’) vs. parallelization over outputs (‘gather’)

- to understand methods by which input parallelism can be transformed to output parallelism

# What Do Scatter and Gather Mean?

In the high-performance literature,

- **scatter** refers to **writing** values **to non-contiguous memory** locations, and

- **gather** refers to **reading** values **from non-contiguous memory** locations.

# Consider a Common Input-Output Computation

The **data**:

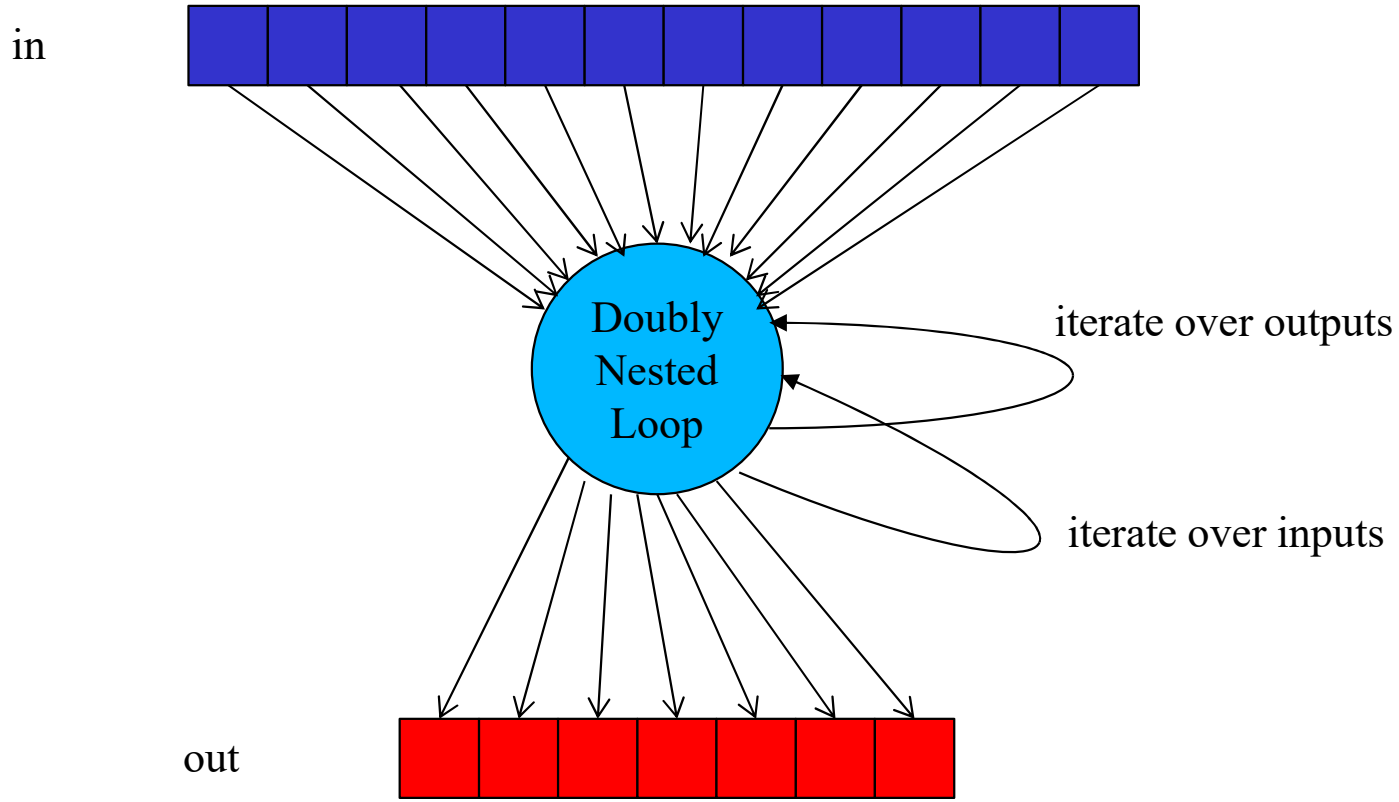
- an **array of inputs**, and
- an **array of outputs**.

The **computation**:

- a (logical) **bipartite graph** from inputs to outputs,
- a **way of calculating** the effect **for each pair**, and
- **an operator**—typically associative and commutative*—that **accumulates results** for each output.

* In real codes, remember that floating-point is not associative.

# Even Simpler: a Complete Bipartite Graph



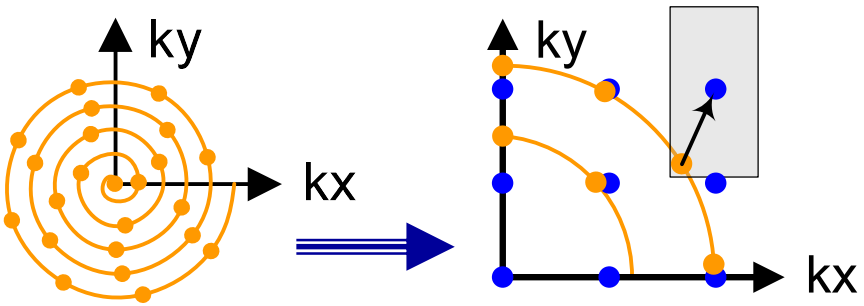

 

# Simple Example: Regularization of MRI Data

```
for (m = 0; m < M; m++) {

  for (n = 0; n < N; n++) {

    out[n] += f(in[m], m, n);
  }
}
```




- **Map** data in **radial** coordinates **to Cartesian** coordinates.
- Input `in`: **M** scan **points**
- Output `out`: **N regularized** scan **points**
- Complexity **O(MN)**.
- **Output** tends to be **more regular than input**.

# Which Loop Should We Transform to Threads?

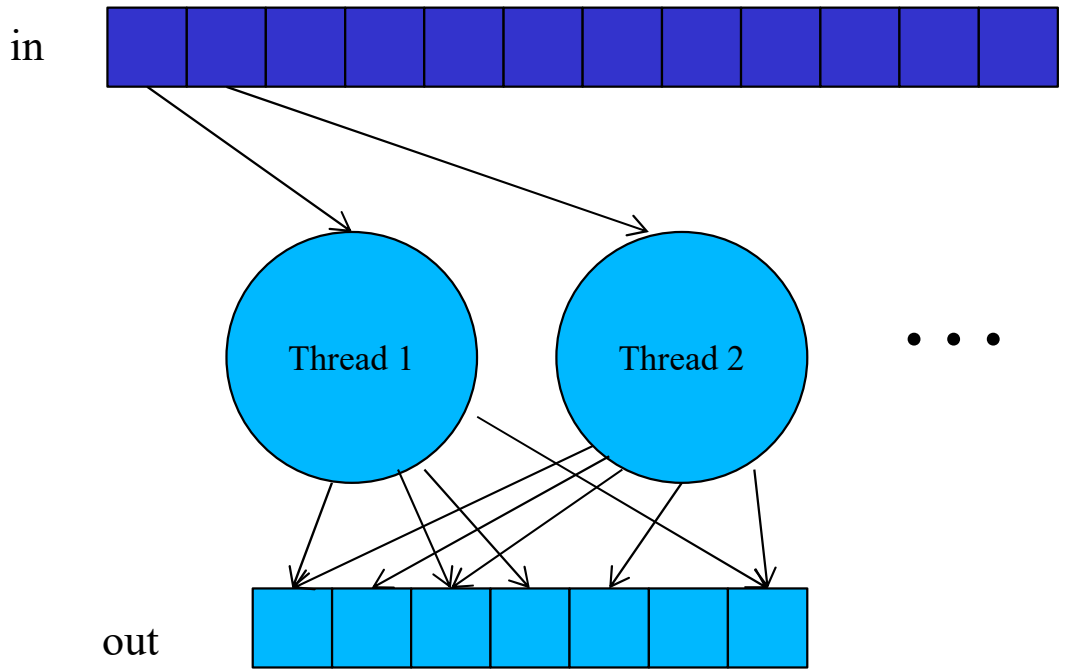
The big question:

**How should we parallelize?**

**Over inputs?**

**Or over outputs?**

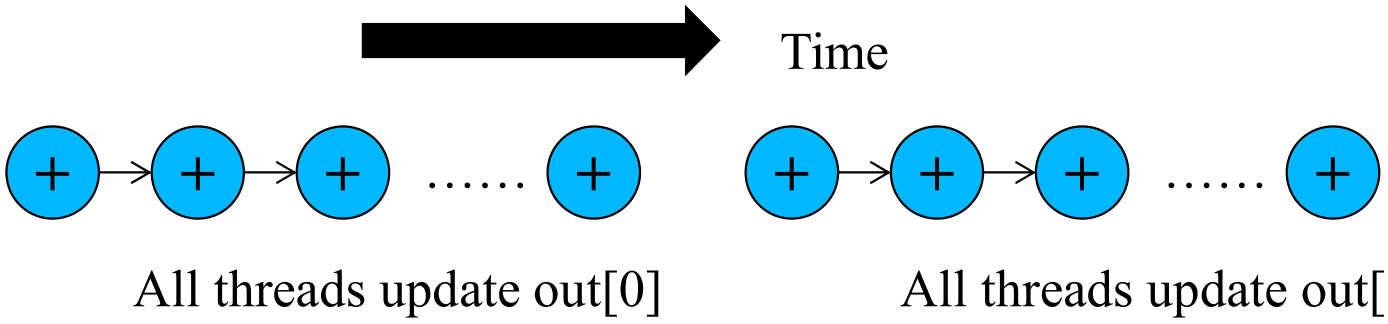
# Parallelization Over Inputs Produces a Scatter



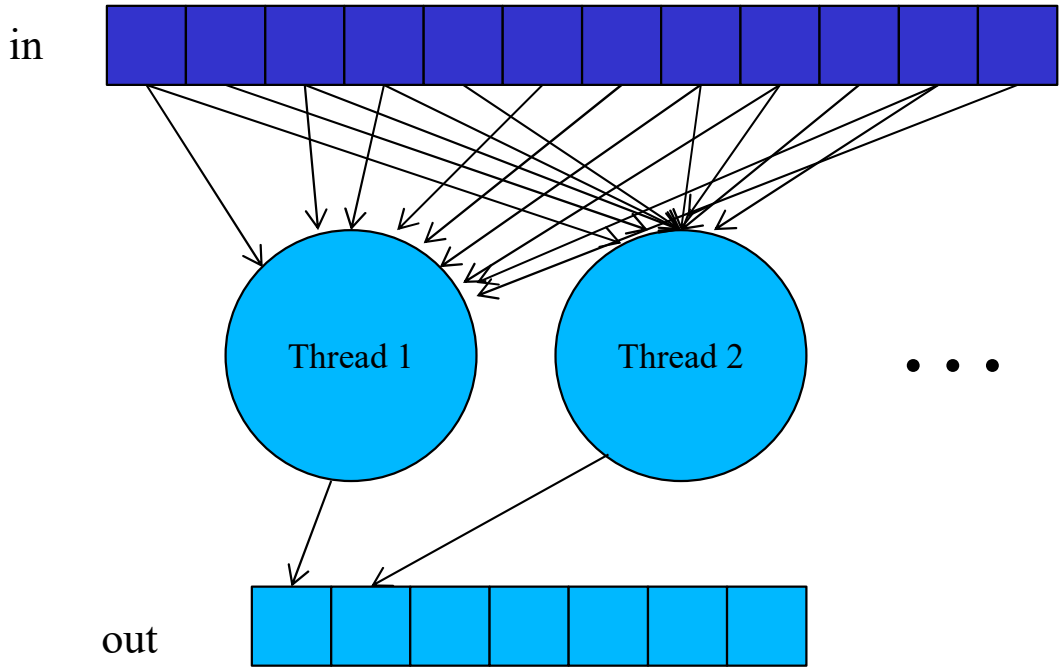
# Scatter is Often Slow on GPUs

- An **output** may be **updated by many** threads.
  - Updates **serialized with atomic** operations.
  - Costly (**slow**) for large numbers of threads.
  - Worse when input-output graph is complete.



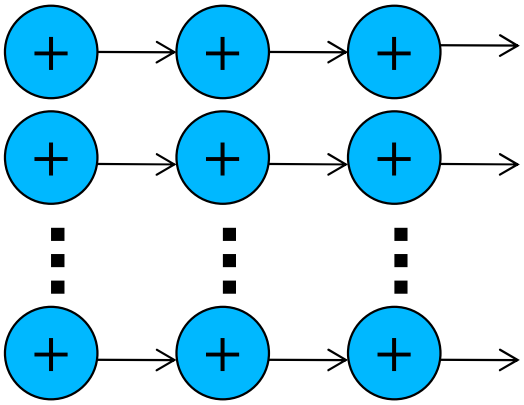

 

# Parallelization Over Outputs Produces a Gather



# Gather Can be Fast on GPUs

- **Avoids serialization!**
  - **Can leverage caches** (**or shared memory**, if graph structure provides locality).
  - Even better when input-output graph is complete (all threads read the same inputs).




• • •

**All threads update their own output elements**; no need for atomics.

# Another Historical Concept: the Owner Computes Rule

Another term for output parallelism

- borrowed from parallel literature
- on programming systems with distributed memory.

Consider a cluster of many processors/chips, each with a memory.

- **Moving computation** from memory to memory **is complicated** and error-prone.
- **Instead**, **move data** to the memory in which an output / result is to be stored, and the **associated processor—the "owner"** of that memory—**computes the result**.
- The term? The "**owner computes**" rule.

# Owner Computes? No, But May Help.

- In GPU community, **output parallelization** is sometimes **referred to as "owner computes."**

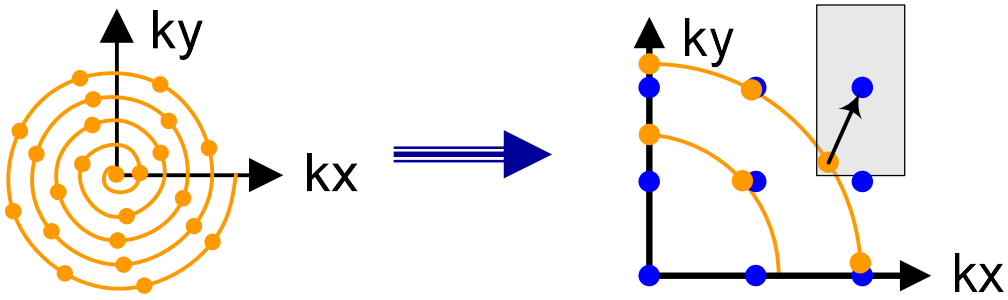
- It **doesn't quite fit**. Memory has no owner.

- However, thinking of the approach in this way may help one
  - remember that only one thread (**choose an owner!**)
  - should access an output
  - **to avoid serialization** (as with parallelization of inputs).

# Computation Often Used to Regularize Data

**So why do any codes parallelize over inputs?**

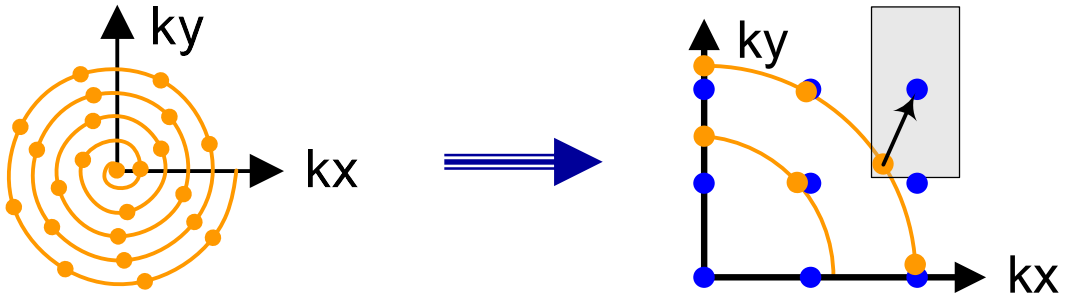
- Many **practical** input-output **graphs**
  - are **sparse** (not complete), and
  - **output tends to be more regular** than input.



# Input-Output Relationship May Make Scatter Easier

**So why do any codes parallelize over inputs?**

- **Calculating outputs** affected by an input is **often easier than calculating inputs** that affect an output.
- Thus **writing a scatter kernel is easier**.



## Today, Assume a Complete Input-Output Relation

One approach that we'll discuss later,

- in the Cut-off Binning Lecture, is to
- regularize input elements to make finding relevant inputs for an output easier.

For this lecture,

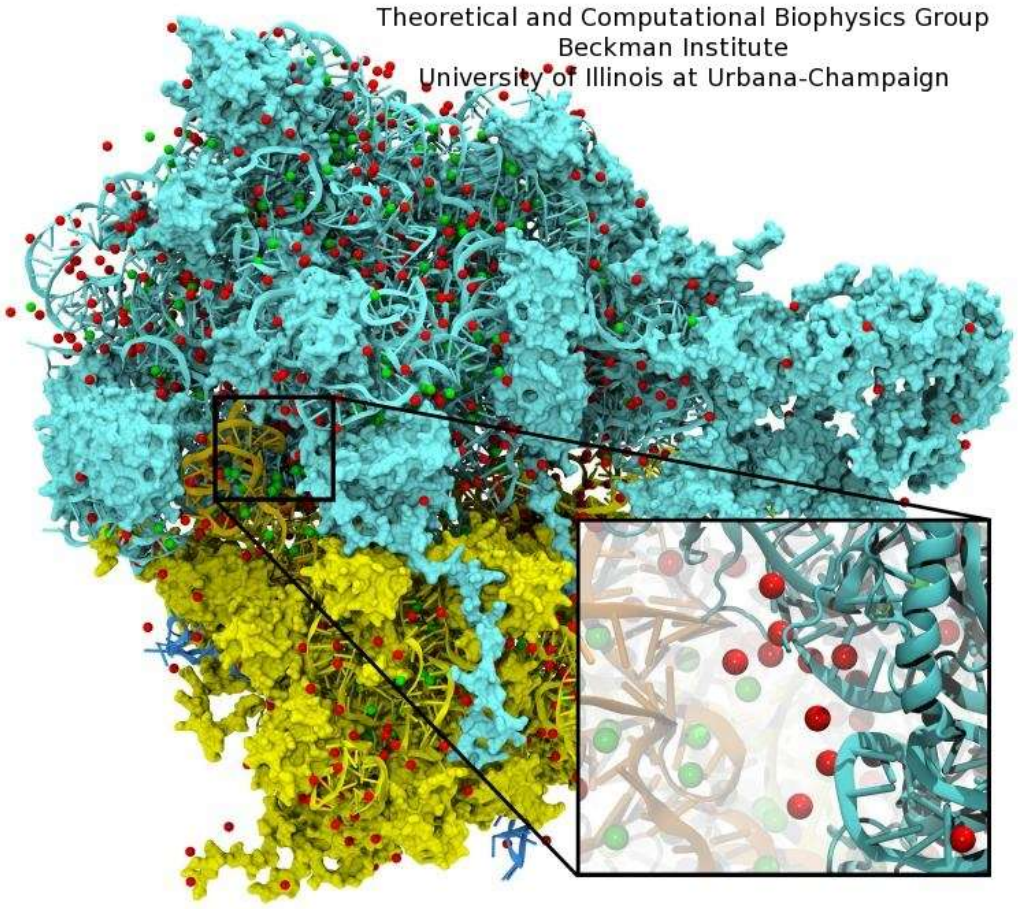
- **assume** a complete bipartite graph.
- In other words, **all inputs effect all outputs**.

# Example: Ion Placement in Molecular Modeling

Let's use an example:

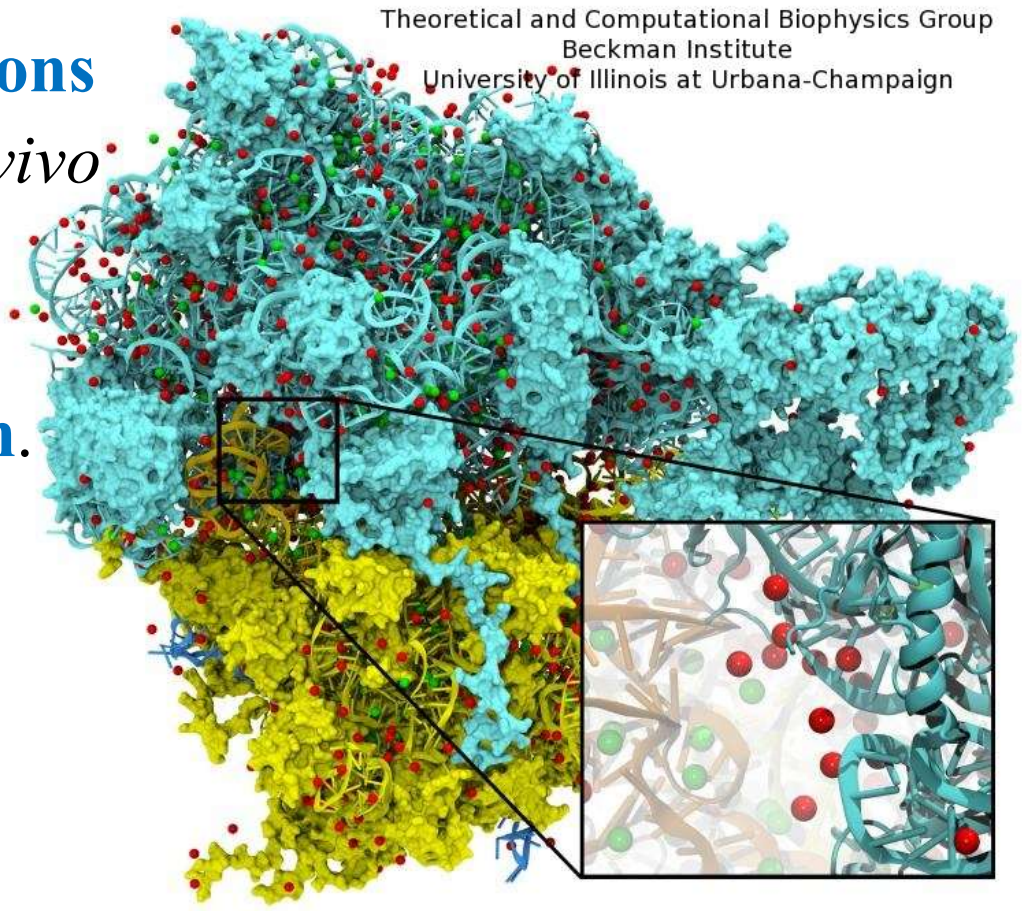
- **ion placement** for molecular modeling
- (drawn from Klaus Schulten's group).



# Ion Placement is Part of Biomolecular Simulation

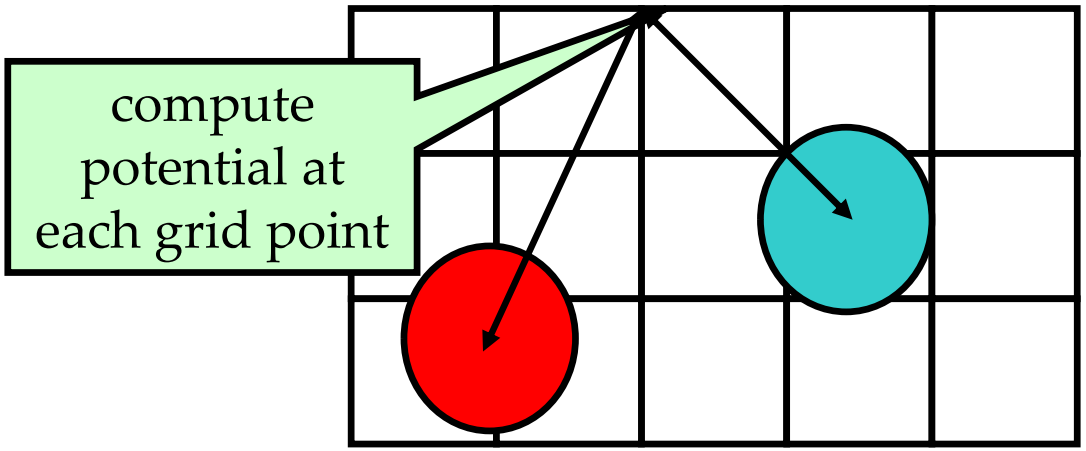
- **Biomolecular simulations** attempt to **replicate** *in vivo* **conditions *in silico***.
- First, **model** structures **constructed in vacuum**.
- Then solvent (water) and **ions added as necessary** to reproduce required biological conditions.



# Calculate Electrostatic Potential Induced by Atoms

The core computation? **Calculate**

- an initial **electrostatic potential** map
- around the simulated structure,
- **considering** the contributions of **all atoms**.




Consumes significant time in VMD, the source of our example.

# Today's Example is Not the Entire Application

We'll ignore this part, but…

- Ions are placed one at a time:
  - find grid point with minimum potential,
  - add an ion at that point,
  - add potential contribution of new ion to map,
  - and repeat until required number of ions added.

# We Explore Direct Coulomb Summation (DCS)

**How do we find electrostatic potential at each grid point?**

Remember basic electromagnetics:

*for each grid/lattice point P and atom A*

*potential[P] += charge[A] / distance (A,P)*

All atoms affect all grid points.

**Most accurate** method: Direct Coulomb Summation (**DCS**)

- **sum all pairwise** (atom to grid point) **potentials**
- (a complete bipartite graph computation!),
- **ideally suited to GPUs**.

# DCS is Not the Fastest Algorithm

Note: approximation-based methods

- such as cut-off summation
- can achieve much higher performance
- at the cost of numerical accuracy and flexibility.

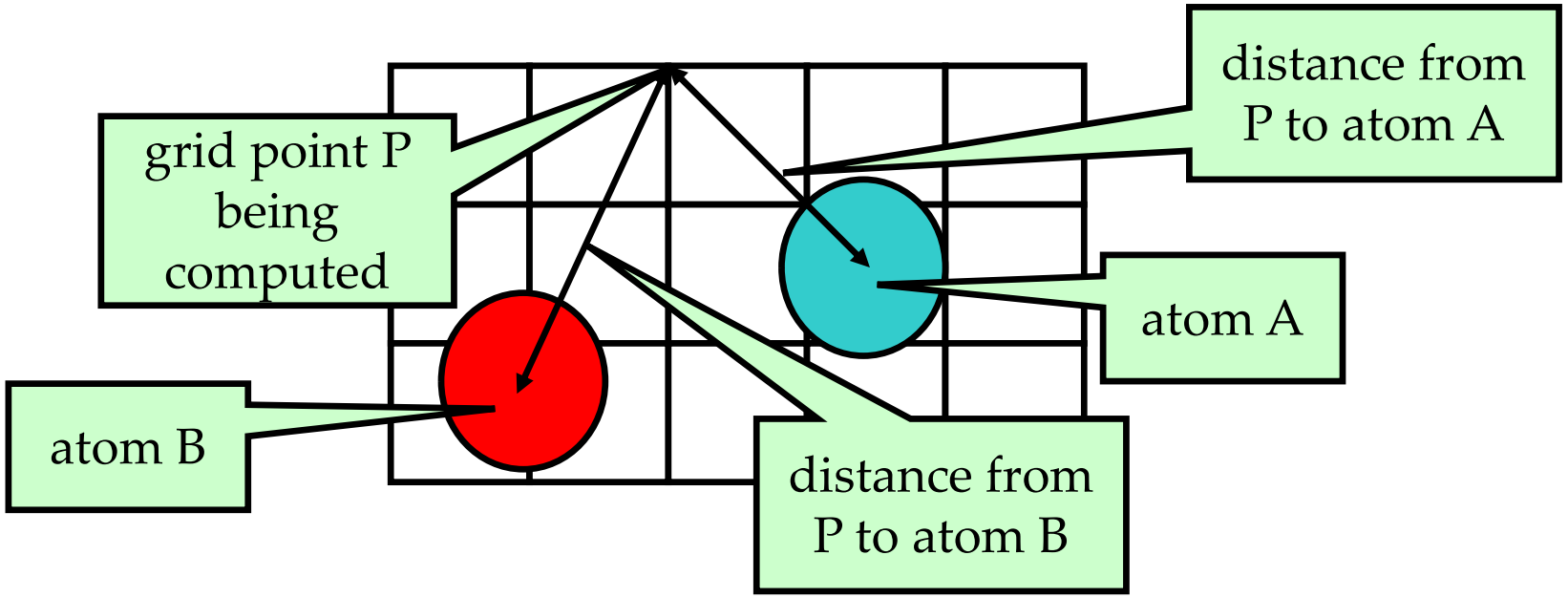
We will cover these later (in the Binning lecture).

# Visualize the DCS Computation

Let's visualize the update rule:

*potential[P] += charge[A] / distance (A,P)*



# Interface to Our DCS Function

**What's the function interface?**

Given

- grid potential (logically a 3D array; the output),
- grid size and spacing, and
- array of atom positions and charges,

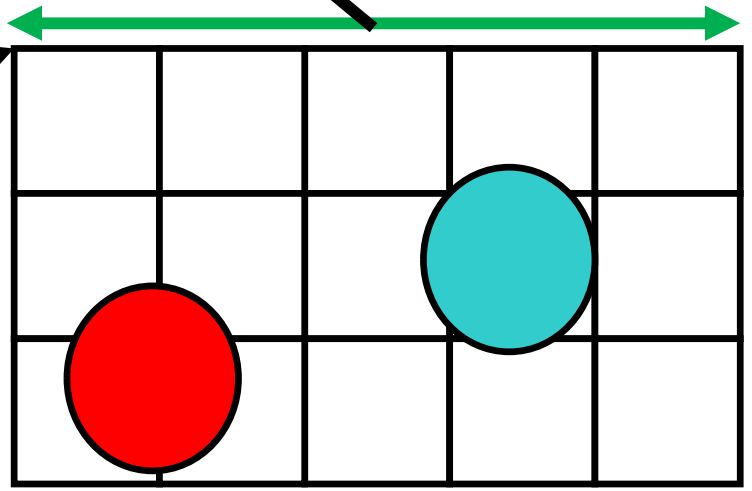
we'll write a function to compute one X-Y slice (a fixed Z coordinate) of the potential.

# Connect the Function to the Visualization

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

pointer to 3D (logical) array of electrostatic potential (the output data)

X, Y, and Z dimensions of the grid



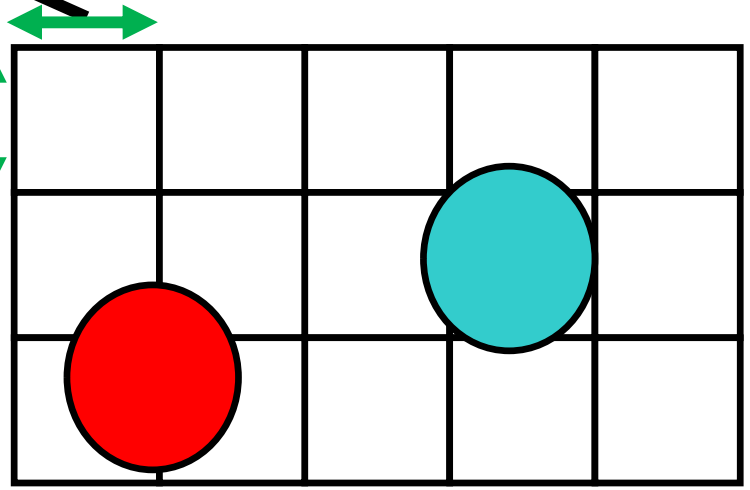
# Connect the Function to the Visualization

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

modeled physical distance between grid points

spatial Z coordinate of X-Y grid slice to compute

Note: spatial Z coordinate is logical Z coordinate times **`gridspacing`**.

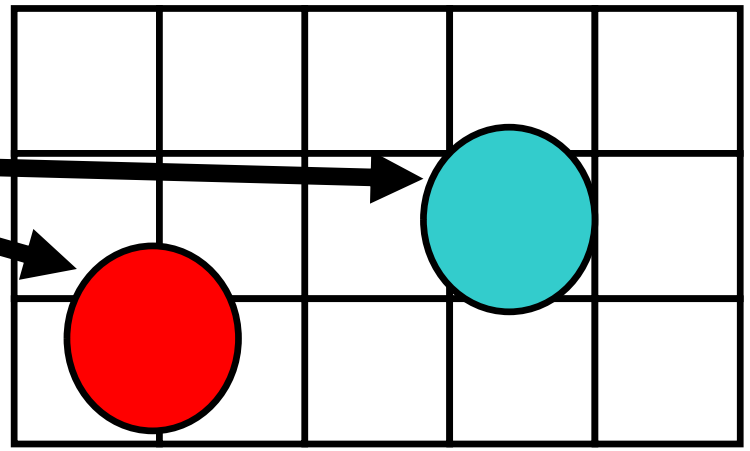


# Connect the Function to the Visualization

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

1D array of X, Y, Z coordinates and charge of atoms (4-tuples, in order stated)

number of atoms in array



# First, Calculate Constants

**Let's start by looking at intuitive sequential version of DCS.**

```
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;

}
```

Remainder on following slides.

first gridpoint for computed X-Y slice (a constant)

# Outer Loop Iterates Over Atoms

```
int atomarrdim = numatoms * 4;

for (int n = 0; n < atomarrdim; n += 4){

}
```

Outer loop iterates over atoms.

Each “atom” is a 4-tuple of floats: X, Y, Z, and charge.

Loop body on following slides.

# Precalculate Values Constant Over Gridpoints

**Why iterate first over atoms?**

```
float dz = z - atoms[n + 2];
float dz2 = dz * dz;
```

All grid points have the same Z value.

```
float charge = atoms[n + 3];
```

Atom's charge is the same for all grid points.

Next, loop over gridpoints.

# Precalculate Values Constant Over Row

**Next, loop over rows (Y dimension).**

```
for (int j = 0; j < grid.y; j++) {
    float y = gridspacing * (float)j;
    float dy = y - atoms[n + 1];
    float dy2 = dy * dy;

    int grid_row_offset =
        grid_slice_offset + grid.x * j;

    Loop across columns here.
}
```

All grid points in row have same Y value.

first gridpoint for row

# Precalculate Values Constant Over Row

**Finally, loop over columns (X dimension).**

```
for (int i = 0; i < grid.x; i++) {
    float x = gridspacing * (float)i;
    float dx = x - atoms[n + 0];

    energygrid[grid_row_offset + i] +=
        charge / sqrtf (dx * dx + dy2 + dz2);
}
```

Compute X value and distance.

Apply update rule (charge over distance).

# Reference Version: Intuitive Sequential DCS

```
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 + 0];
         energygrid[grid_row_offset + i] += charge / sqrtf (dx * dx + dy2 + dz2);
      }
    }
  }
}
```

# Summary of Sequential DCS

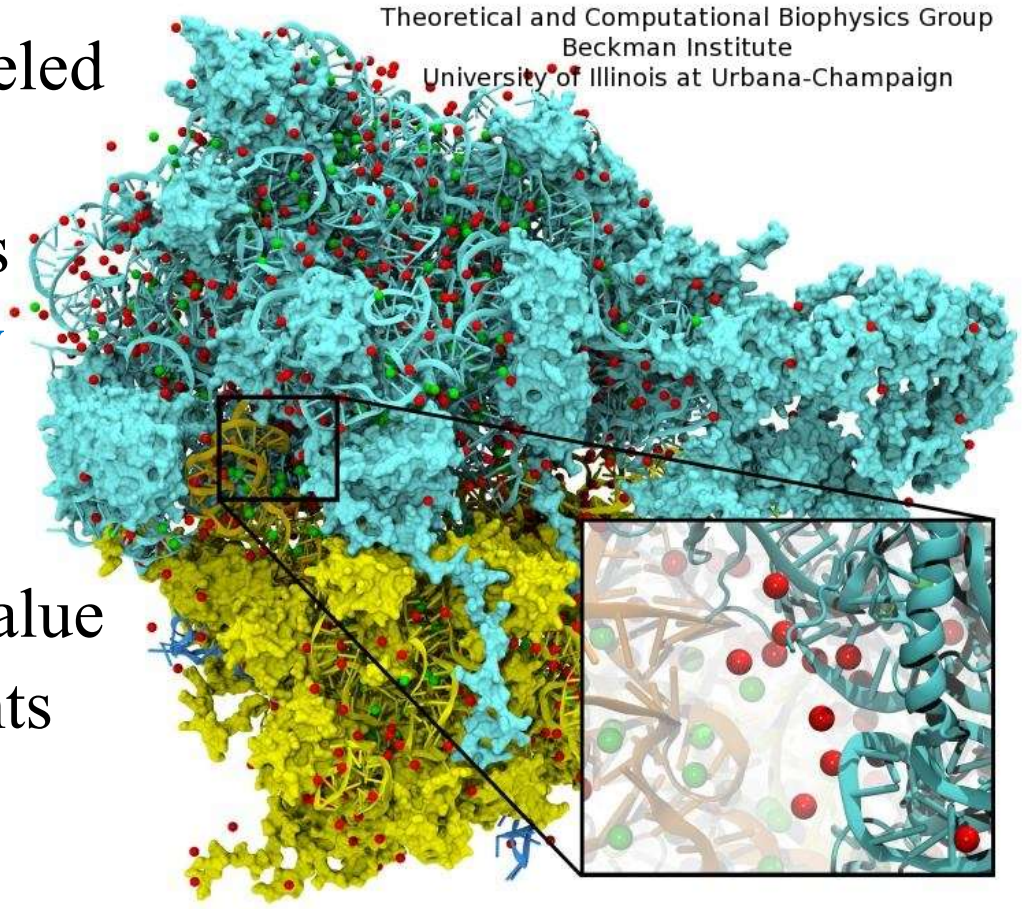
- **Input-oriented**: for each input atom, calculate potential contribution for all gridpoints in X-Y slice.
- **Output** (`energygrid`)
  - **is** a **regular** lattice, with
  - a linear mapping from indices to spatial coordinates.
- **Input** (atom positions) **is irregular**: spatial coordinates of atoms (X, Y, Z) stored as data in `atom`.
- Algorithm **minimizes computation** by moving invariants out of loops and pre-computing partial expressions.

# 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**



# Create a Simple CUDA Version as Usual

**Recall the canonical process for producing a simple CUDA kernel.**

1. Input atom coordinates and charges to host memory.
2. Allocate and initialize host copy memory of potential map.
3. Allocate potential map slice buffer on GPU.
4. Loop over potential map slices:
   a. Copy potential map slice from host to GPU.
   b. Loop over groups of atoms:
      i. Copy atom data to GPU.
      ii. Run CUDA Kernel on atoms and potential map slice on GPU.
   c. Copy potential map slice from GPU to host.
5. Free resources.

# Straightforward CUDA Parallelization

- **Parallelize the outer loop** (over atoms).
  - Each **thread computes**
    - the **contribution of** an **atom**
    - **to all grid points** in the X-Y slice.
  - **Scatter parallelization**.
- **Strip outer loop** from CPU version **to obtain kernel**.
  - Each thread executes an outer loop iteration
  - (plus redundant constant computation).
- **`numatoms` used in** host code to configure **kernel launch**.

# From Sequential to Kernel

```
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 + 0];
         energygrid[grid_row_offset + i] += charge / sqrtf (dx * dx + dy2 + dz2);
      }
    }
  }
}
```

Replace with calculation of n from thread indices.

# DCS Scatter Kernel (Slow!)

```
void __global__ cenergy (float* energygrid, dim3 grid, float gridspacing, float z,
                         const float* atoms /* numatoms not passed */) {
  int n = (blockIdx.x * blockDim.x + threadIdx.x) * 4;
  int grid_slice_offset = (grid.x * grid.y * z) / gridspacing;
  // no loop over atoms
  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 + 0];
       energygrid[grid_row_offset + i] += charge / sqrtf (dx * dx + dy2 + dz2);
    }
  }
  // no loop over atoms
}
```

Oops! Update needs to be atomic!

New code in blue.

# Review Atomic Operations in CUDA

Atomic operations in CUDA

- **look like function calls** syntactically, but
- compiler translates them into **single instructions** (synchronization primitives, or **intrinsics** in CUDA terminology);
- examples include fetch-and-op (add, sub, inc, dec, min, max), swap/exchange, CAS (compare and swap).

(For details: CUDA C programming Guide 4.0+)

# Example of Fetch-and-Add Operation

Example: atomic fetch-and-add

```
int atomicAdd (int* address, int val);
```

- Reads the 32-bit word at **`address`** (global or shared memory) into **OLD**,
- stores (**OLD** + **`val`**) back to memory at address, and
- returns **OLD**.

# Several Fetch-and-Op Data Types Supported

32-bit unsigned:

```
unsigned int atomicAdd
(unsigned int* address, unsigned int val);
```

64-bit unsigned:

```
unsigned long long int atomicAdd (unsigned long
long int* address, unsigned long long int val);
```

single-precision floating-point (capability > 2.0):

```
float atomicAdd
    (float* address, float val);
```

# Pros and Cons of the Scatter Kernel

- Pros
  - **easy to write** based on CPU version
  - **good for** software engineering and code **maintenance**
  - **preserves computation efficiency** (coordinates, distances, offsets) of sequential code

- Cons
  - **atomic add serializes** the execution, so **slow**!

# Again, Calculate Constants First

**Let's examine an output-oriented sequential DCS.**

```
void cenergy (float* energygrid, dim3 grid,
              float gridspacing, float z,
              const float* atoms, int numatoms)
{
  int atomarrdim = numatoms * 4;
  int k = z / gridspacing;

}
```

Remainder on following slides.

size of **atoms** array and logical Z coordinate (constants)

# Outer Loop Iterates Over Rows

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

Outer loop iterates over rows.

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

Spatial Y coordinate is constant for each row.

Remaining loop body on following slides.

```
}
```

# Second Loop Iterates Over Columns

**Next, loop over columns.**

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

  float x = gridspacing * (float)i;
  float energy = 0.0f;

  energygrid[grid.x * grid.y * k + grid.x * j + i]
      += energy;

}
```

Second loop iterates over columns.

Spatial X coordinate is constant for each grid point.

Remaining loop body on following slides.

# Use a Local Variable to Accumulate Contributions

**Local variable accumulates all contributions.**

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

  float x = gridspacing * (float)i;
  float energy = 0.0f;

  energygrid[grid.x * grid.y * k + grid.x * j + i]
      += energy;
}
```

Initialize accumulation.

Remaining loop body on following slides.

Accumulate here.

Add accumulation into grid point.

# Inner Loop Iterates Over Atoms

**Loop over atoms to accumulate contribution.**

```
for (int n = 0; n < atomarrdim; n += 4) {
  float dx = x - atoms[n + 0];
  float dy = y - atoms[n + 1];
  float dz = z - atoms[n + 2];

  energy += atoms[n + 3] /
    sqrtf (dx * dx + dy * dy + dz * dz);
}
```

Inner loop iterates over atoms (4-tuples).

# Accumulate Contribution from Each Atom

**Compute distance and accumulate atom's contribution.**

```
for (int n = 0; n < atomarrdim; n += 4) {
   float dx = x - atoms[n + 0];
   float dy = y - atoms[n + 1];
   float dz = z - atoms[n + 2];

   energy += atoms[n + 3] /
     sqrtf (dx * dx + dy * dy + dz * dz);
}
```

Compute per-atom distance components.

Accumulate contribution (charge over distance).

# Reference Version: Output-Oriented Sequential DCS

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

# Pros and Cons of the Output-Oriented Sequential DCS

- Pros
  - **fewer accesses to `energygrid` array**
  - **simpler code** structure

- Cons
  - **more calculations** on the coordinates.
  - **more accesses to `atom` array**
  - **slower execution** (due to extra calculations)

# Parallelize New Version with Two Loops

- **Parallelize two outer loops** (over grid points).
  - Each **thread computes**
    - the **contribution to one grid point** in the X-Y slice
    - **from all atoms**.
  - **Gather parallelization**.
- **Strip outer loops** from CPU version **to obtain kernel**.
  - Each thread executes a loop iteration
  - (plus redundant constant computation).

# From Output-Oriented Sequential to Kernel

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

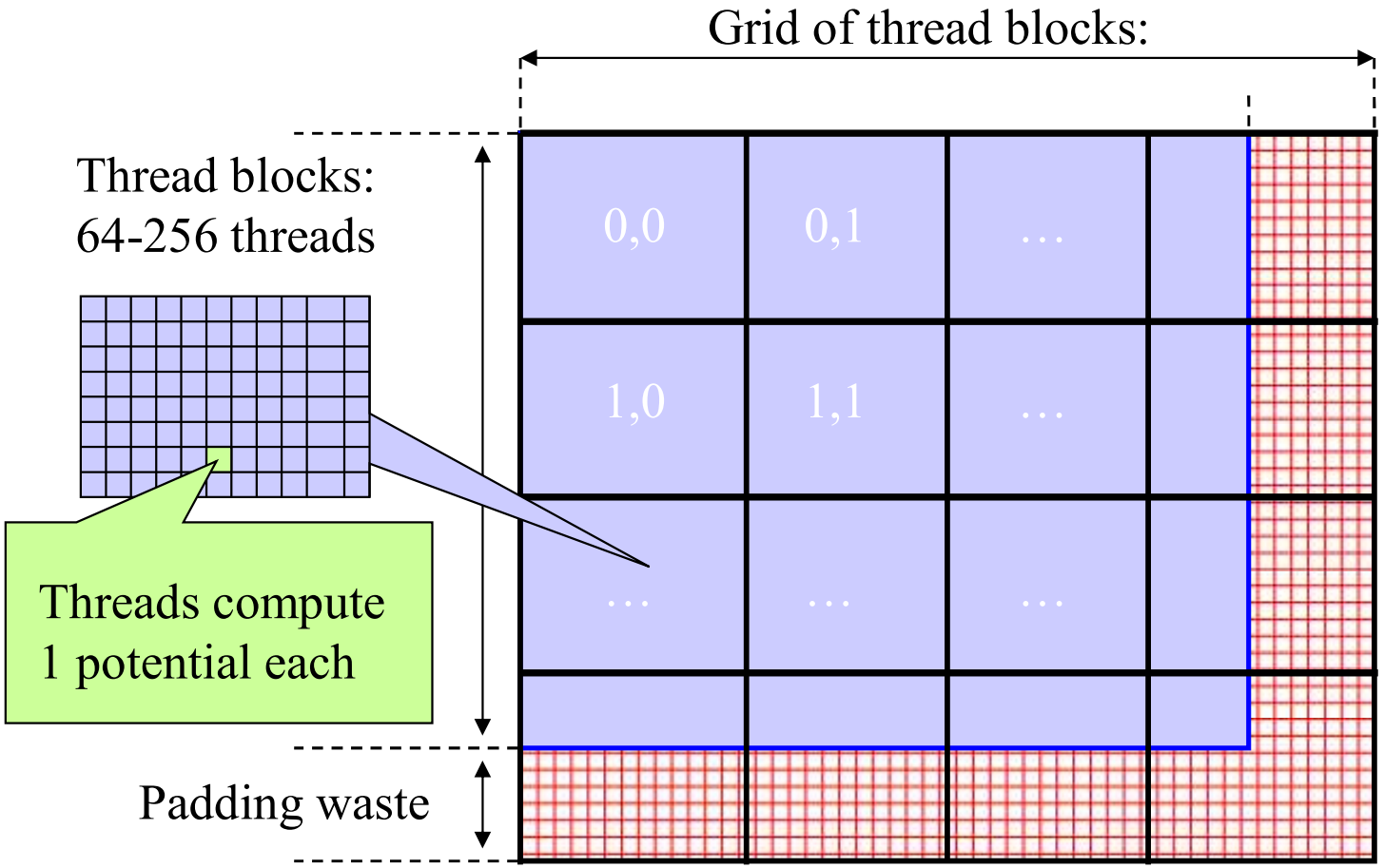
Replace with calculation of i, j from thread indices.

# DCS Gather Kernel (Fast!)

```
void __global__ cenergy (float* energygrid, dim3 grid, float gridspacing, float z,
                         const 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;
  // no loop over rows
  float y = gridspacing * (float)j;
  // no loop over columns
  float x = gridspacing * (float)i;
  float energy = 0.0f;
  for (int n = 0; n < atomarrdim; n += 4){   // Calculate contribution of each atom.
    float dx = x - atoms[n + 0];
    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;
  // no loop over columns
  // no loop over rows
}
```

New code in blue.

# Block/Grid Decomposition (no thread coarsening)



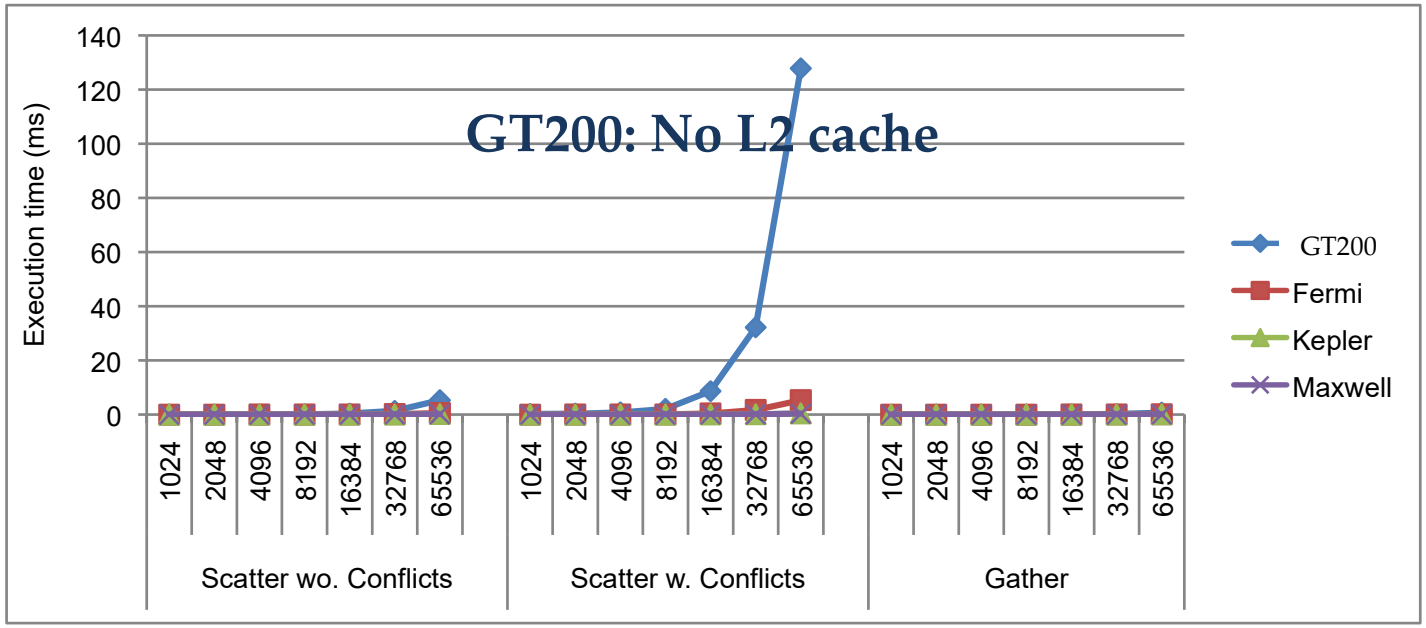
# Compare Gather Kernel with Scatter Kernel

- Threads
  - do not require atomic operations (**no serialization**),
  - **read** same **`atom`** array values **in** the **same order**, and
  - read/**write unique `energygrid` value** at end of thread.
- Optionally, **`dz*dz`** can be sent in place of **`z`**.
- Implications
  - **Gather** kernel is **much faster than scatter** kernel!
  - **Compute-efficient sequential** algorithm **may not translate to fastest parallel** kernel (gather vs. scatter).
  - We will return to this point later.

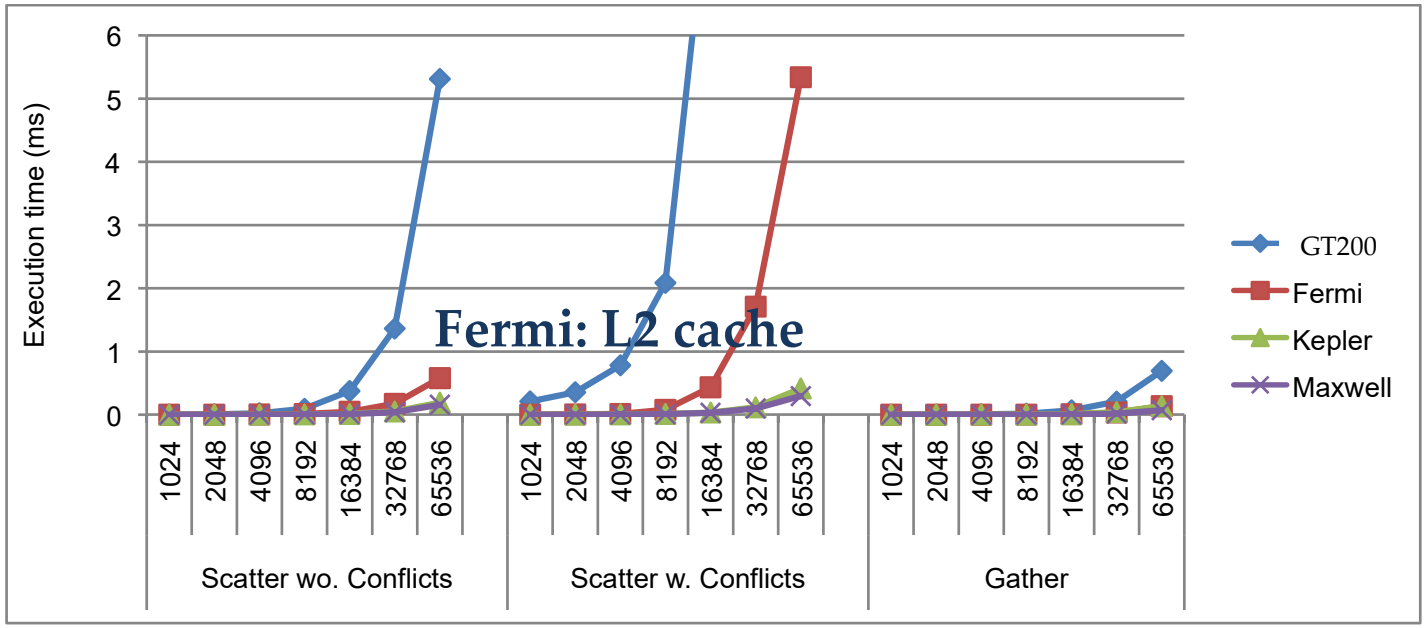
# Case study: Scatter vs. Gather

- Scatter vs. Gather: most data negligible at this scale



# Case study: Scatter vs. Gather

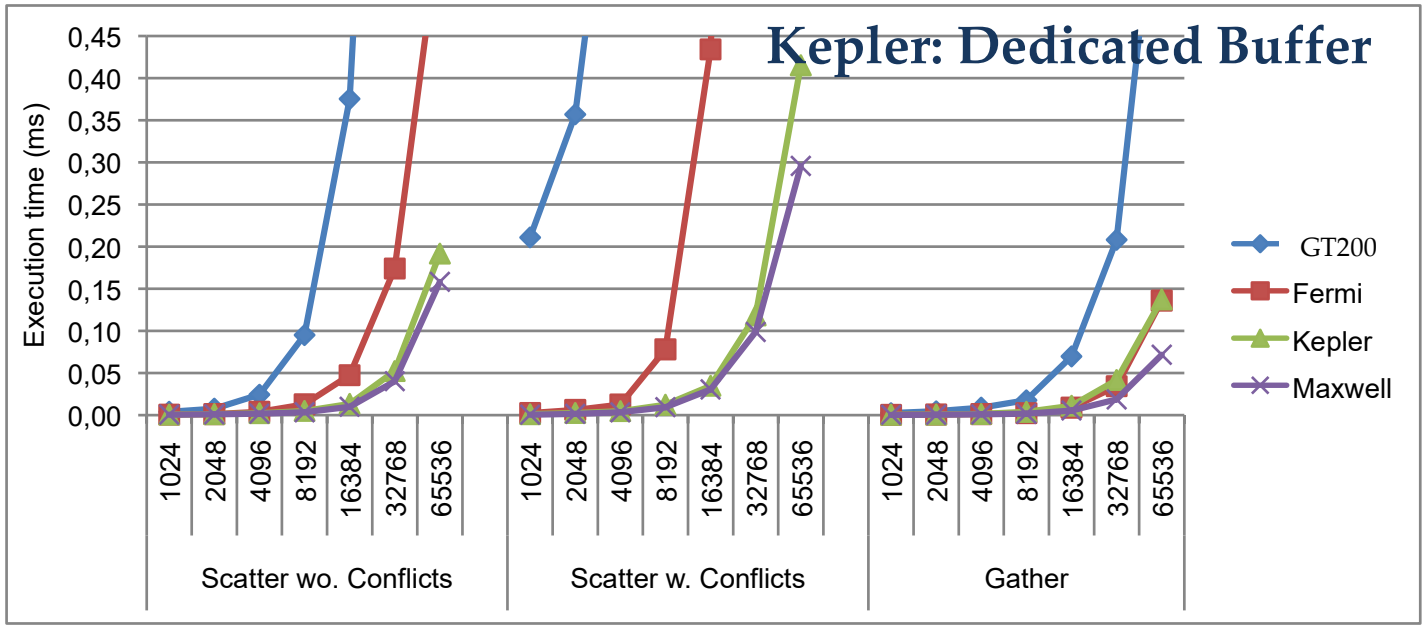
- Zoom in vertical axis by 23×



# Case study: Scatter vs. gather

- Zoom in vertical axis another 13.3×



# Cache Performance also Dominates CPU Timing

- In modern **CPUs**,
  - **cache** effectiveness often **more important**
  - **than compute** efficiency.
- "Intuitive" sequential **DCS (scatter)**
  **has bad cache performance**
  - **`energygrid`** is large, typically **>20×** larger than **`atom`**
  - **Code** sweeps through **`energygrid`**
    for each atom, **trashing cache**.
- **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 (x,y,z) grid point
            using precomputed dy^2 and dz^2
      }
    }
  }
}
```

Use additional, pre-computed arrays.

Extra arrays? So …
Why better cache behavior?

# More Thoughts on Fast Sequential Code

**Why does this code have better cache behavior on CPUs?**

- Recall that **`atom`** is **much smaller than `energygrid`**.
- Sweeping **`atom`** repeatedly leverages the cache.
- Even several copies (the extra arrays) can fit!
  (Read: **atom group size chosen based on cache** size.)

**The lesson: Writing high performance code** of any type (sequential or parallel) **is** an **engineering** design effort. **Tradeoffs** often **depend on data** sizes.

# What About Other Ways of Parallelizing?

**Other choices are possible for parallelization…**

In fact, those who have taken 408 have implemented some.

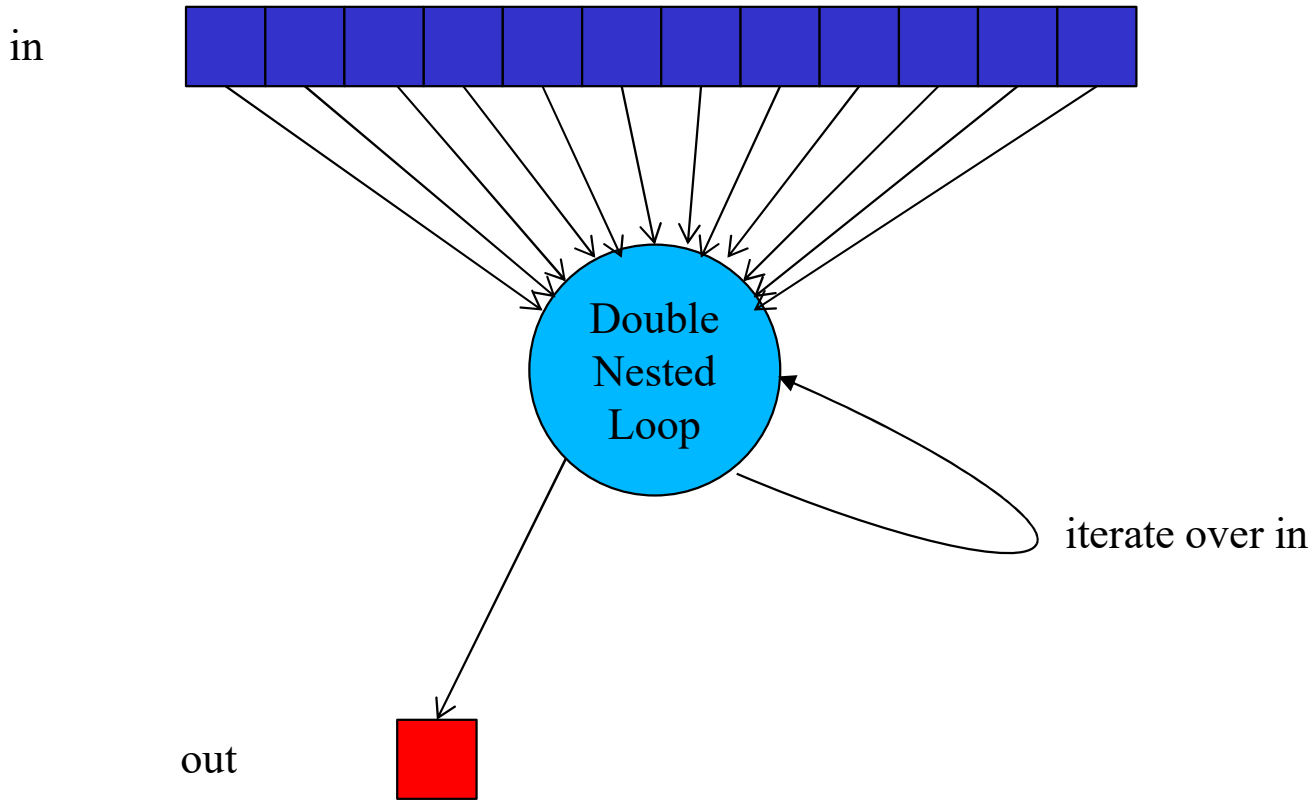
**Consider reduction.**

**One output.**

**Complete bipartite graph**

(all inputs affect the output)!

# A Reduction Pattern



# Reduction Has No Output Parallelism!

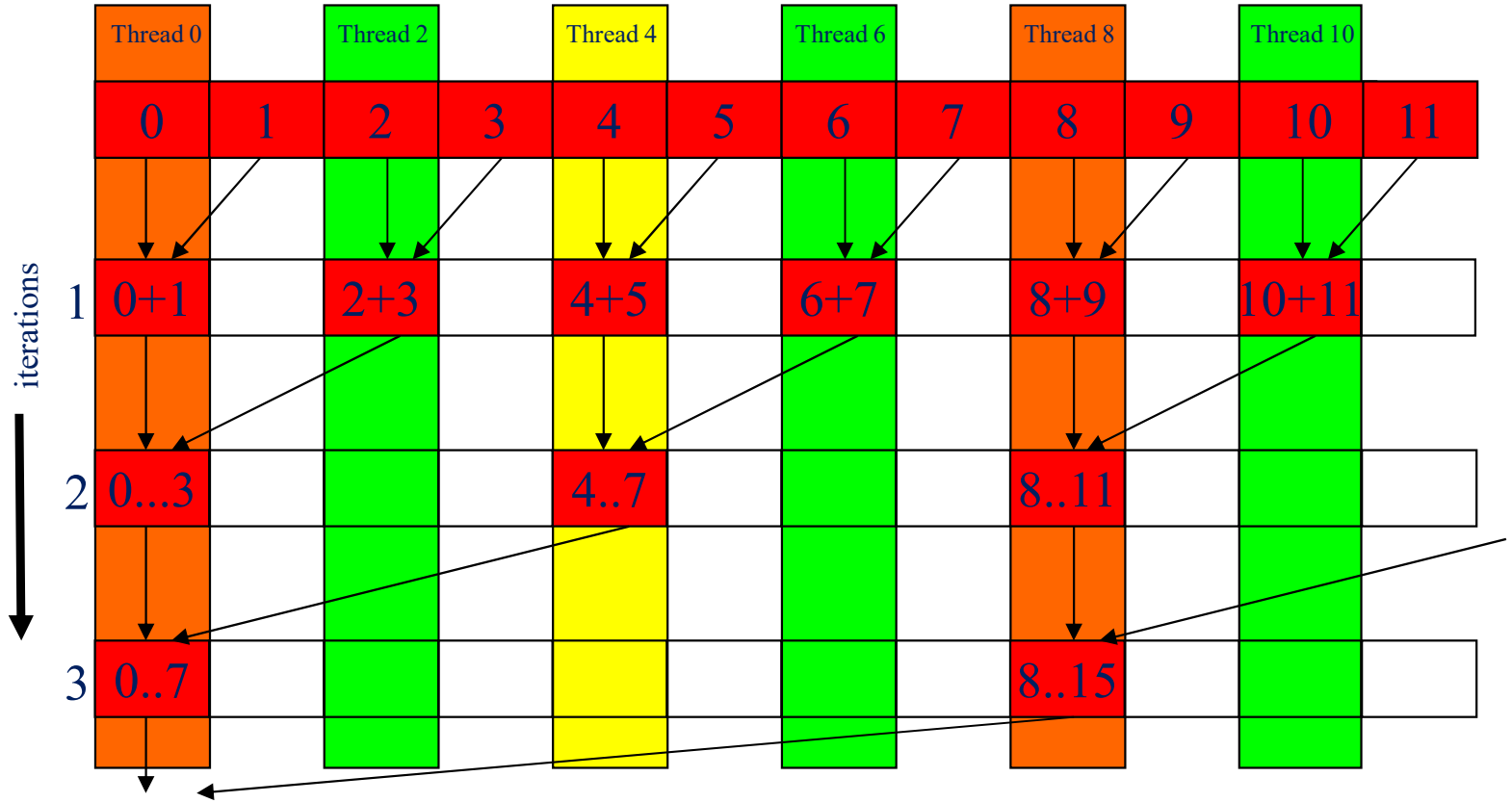
- Output parallelization … one thread?

- But **scatter-style** code is **not acceptable**:
  - each thread reads one input and accumulates into one reduction variable with atomic operation, so
  - ALL input threads write to ONE output location?

- **Privatization or Tree reduction** makes more sense.

# Solution: Create Multiple Outputs



# ANY QUESTIONS?