ECE 508
Manycore Parallel Algorithms

Lecture 6: Parallel/Compact Binning

Objective

- to learn to parallelize binning, and
- to learn techniques for compacting binned data for
  - better utilization of on-chip memory and
  - reduction of data transfer volume from global to on-chip memory

Binning on CPU Can Be a Problem

With many atoms, binning is slow.

Remember Amdahl's law?

Sometimes we need to parallelize binning.

Obvious Approach Requires a Scatter

The challenge: binning is fundamentally a scatter.

- Atom locations are irregular.
- Eliminating conflicts can require
  - significant extra work and/or
  - narrow parallelism.
- A gather pattern, for example, is quadratic.

We can't use the bins until we bin the atoms…

Binning is Much Like Computing a Histogram

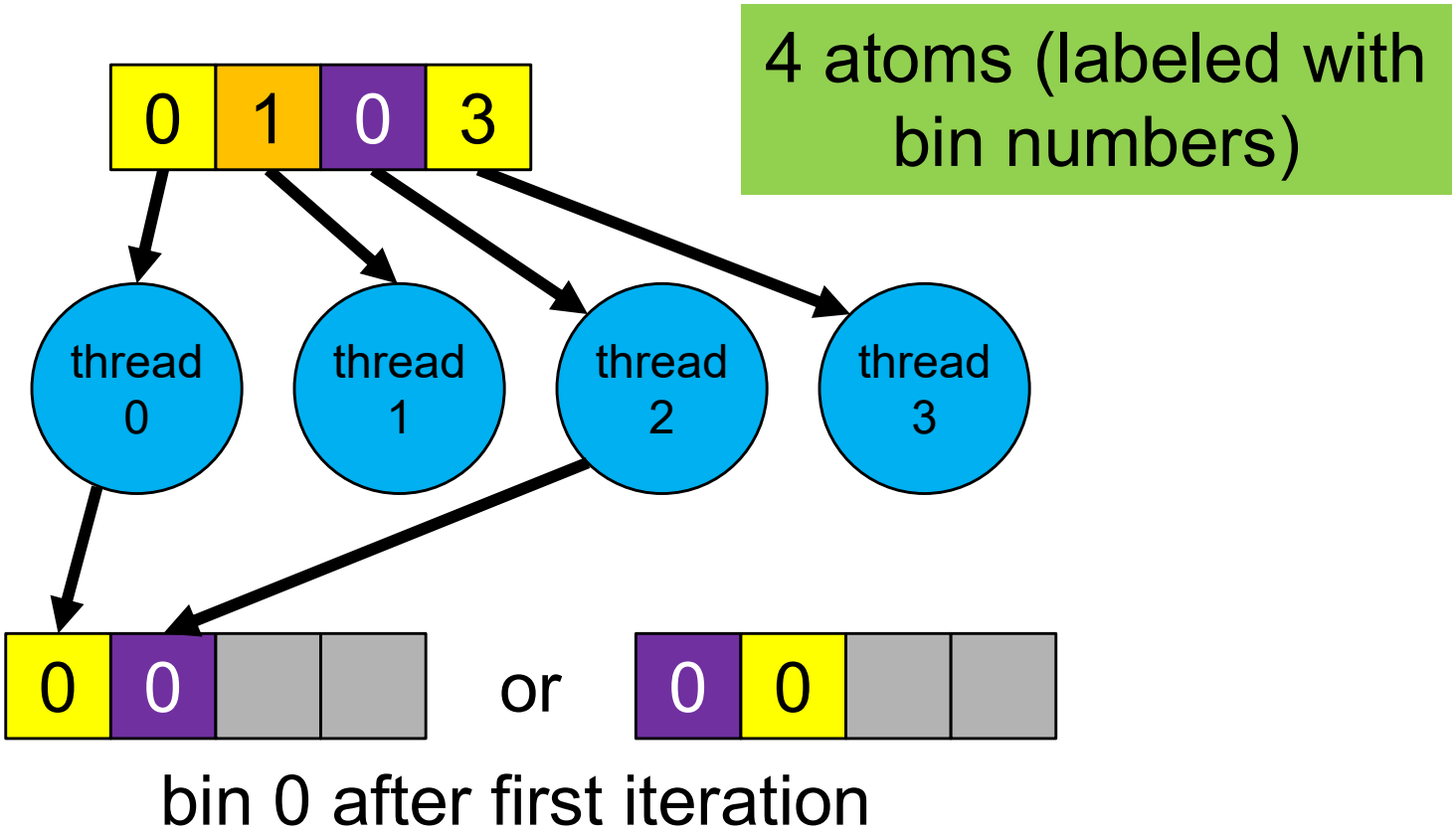
Binning is not an **all-to-all** relationship.

- **Each input** (atom) **goes to** only **one output** (bin),
- **so conflicts are manageable**.

Pattern similar to a familiar problem

- from 408 (or equivalent introductory GPU course):
- **computing a histogram**.

Illustration of First Histogram Step



Atomic Operations Identify Final Order

With **one small difference**:

- to put each atom into memory
  (into the appropriate bin),
- we **need per-atom indices**
  (the result of the atomic fetch-and-adds).

The per-atom indices are unique, so we can **copy each atom's data without conflict**.

Two-Step Process for Binning

Conceptually, we have the following:

1. **Compute histogram**
   - of number of atoms in each bin, and
   - save per-atom indices.
2. Use per-atom indices
   - to **copy** each atom's **data**
   - **to** its **assigned location** in a bin.

Merge Both Steps into One Kernel

Practically,

- per-atom **indices used only in Step 2**,
- so why **write and reload from global memory**?

One answer:

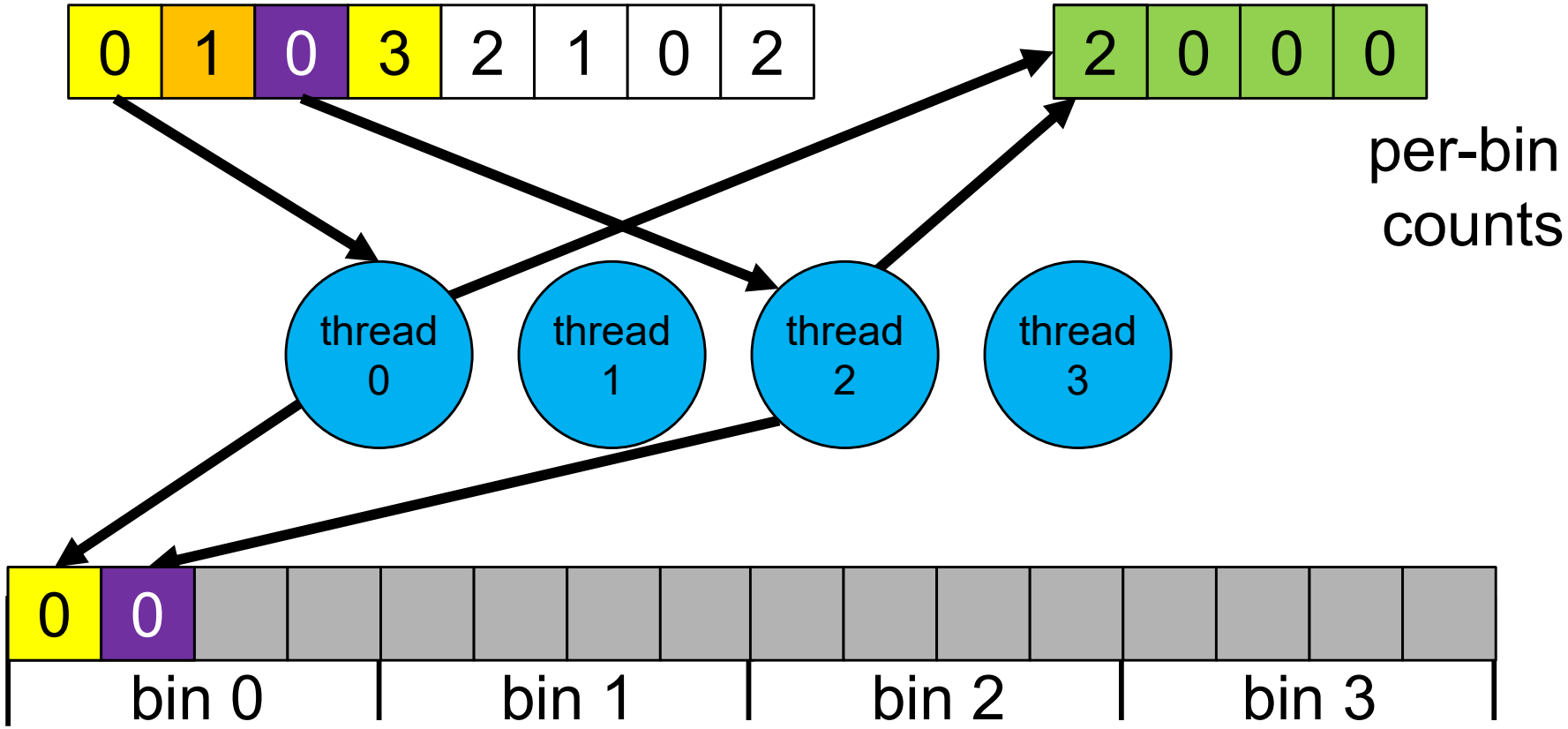
- software engineering / code reusability, but
- at odds with efficiency!

We'll **merge these two steps** into one—called **kernel fusion** if the codes were written separately.

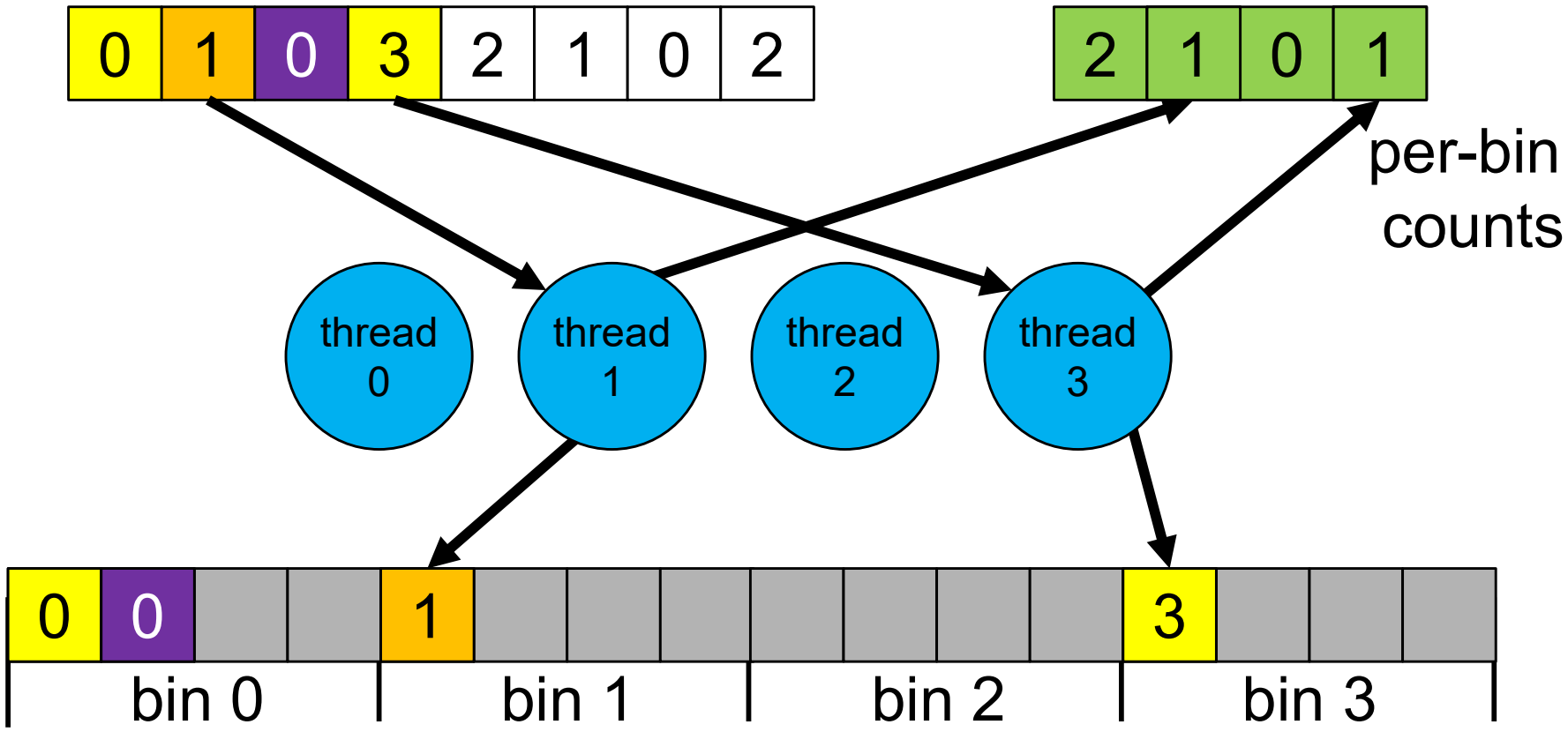
Pseudo-Code for a Parallel Binning Kernel

```
for each input element Elem {
  BI ← bin index for Elem
  atom_idx ← atomicAdd (&counters[BI], 1)
  insert atom into bin[BI][atom_idx]
}
```

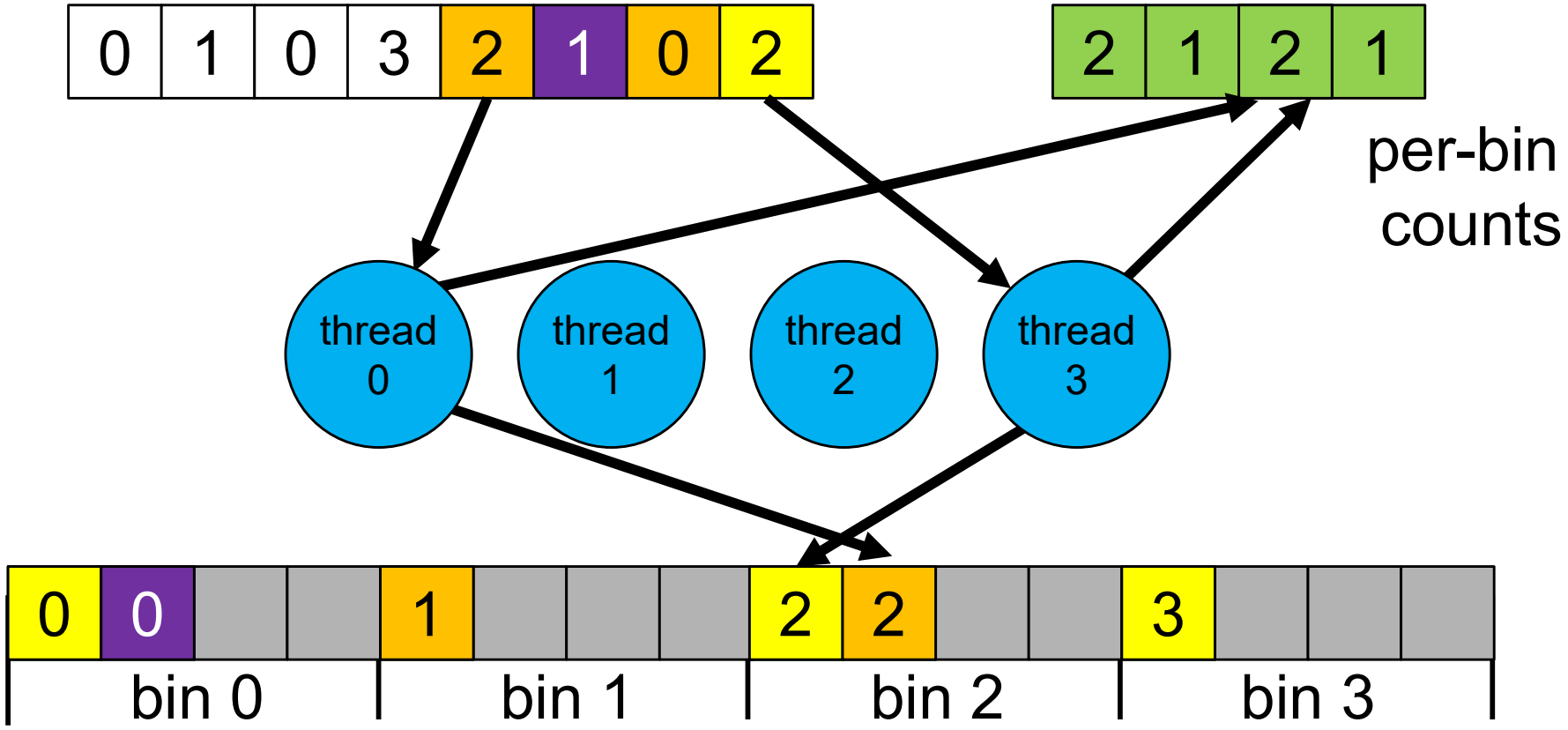
First Binning Iteration, Threads 0 and 2



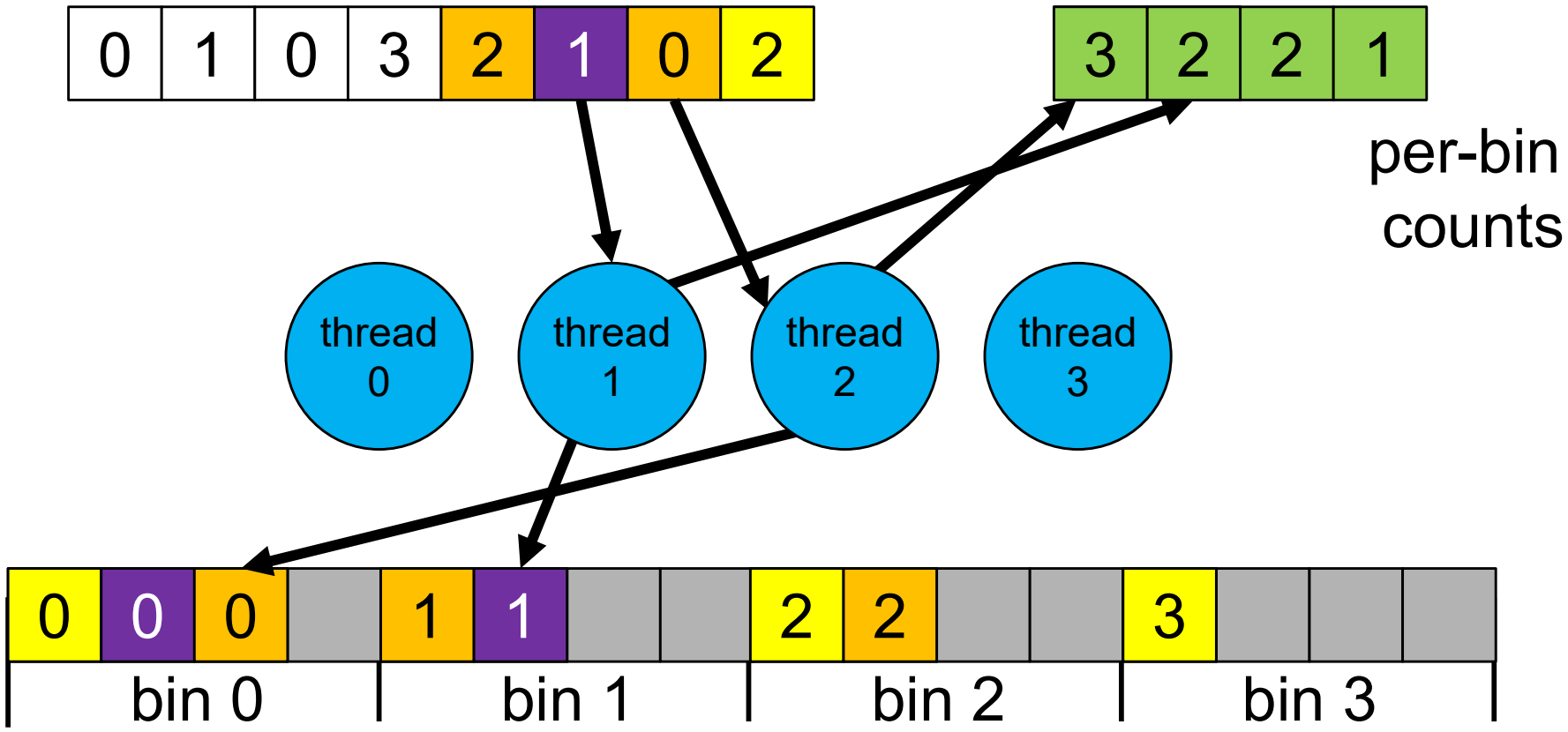
Simultaneously, Threads 1 and 3



Second Binning Iteration, Threads 0 and 3



Simultaneously, Threads 1 and 2



Comments on Parallel Binning Kernel

- **Bin number is abstract**:
  - could be 1D, as in Lab 4, or
  - could be linearized from 2D or 3D.
- **Order of atoms within bins** is **arbitrary**, and decided by serialization of atomics to bin's counter.
- **Writes to bins**
  - do not coalesce well, but
  - **latency hidden behind** atomic **conflicts**.

Overflow Bin is Also Needed

Oops! Bin capacity is fixed!

What about overflow?

If per-atom index exceeds capacity,

- atomically **decrement bin count**, then
- **compete for overflow bin** placement.

Fill the GPU with Threads, and No More

How do we launch histogram kernels?

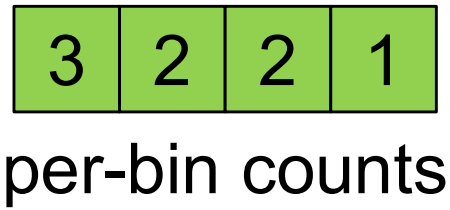
No point fighting for resources!

- Enough thread blocks to keep SMs busy.
- **Threads walk** through data set
  - with **stride `gridDim.x * blockDim.x`**
  - until done.
- **Can privatize** with shared memory, if all bins fit.

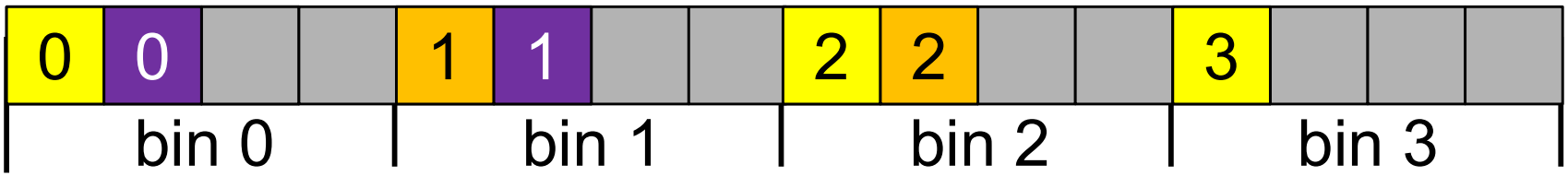
After Binning, Execute Our Cutoff Kernel

After binning, we have

- bins as before, plus
- bin counts.



(Examining count is a global memory read, so not so useful for our cutoff kernel.)



Pseudo-Code for Electrostatic Cutoff Computation

```
// 1. binning
for each atom in the simulation volume,
    index_of_bin := func(atom.addr / BIN_SIZE)
    bin[index_of_bin] += atom
```

GPU

in parallel

```
// 2. generate the neighborhood offset list
for each c from -cutoff to cutoff in all three dimensions,
    if distance(0, c) < cutoff,
        nlist += c
```

CPU

```
// 3. do the computation
for each point in the output grid,
    index_of_bin := point.addr / BIN_SIZE
    for each offset in nlist,
        for each atom in bin[index_of_bin + offset],
            if (within cut-off)
                point.potential += atom.charge / (distance from point to atom)
```

GPU

Reads entire bins into shared memory.

(Simplified) GPU Kernel Inner Loop

Exit when an empty atom bin entry is encountered

Compute dx and dz once

Cylinder test

Four times dy, distance, and cutoff

Cutoff test and potential value calculation

```
for (i = 0;  i < BIN_DEPTH;  i++) {
  aq = AtomBinCache[i].w;
  if (aq == 0) break;

  dx = AtomBinCache[i].x - x;
  dz = AtomBinCache[i].z - z;
  dxdz2 = dx*dx + dz*dz;
  if (dxdz2 < cutoff2) continue;

  dy = AtomBinCache[i].y - y;
  r2 = dy*dy + dxdz2;
  if (r2 < cutoff2)
   /* Simplified example */
    poten0 += aq * rsqrtf(r2);

  dy = dy - grid_spacing;
  /* Repeat three more times */
```

Stop on first dummy atom.

Dummy Atoms Lead to Inefficiencies

Another problem:

dummy atoms cause inefficiency.

Especially for non-uniform atom distributions.

Resources used for dummy atoms in bins:

- global **memory bandwidth**,
- **shared memory** storage, and
- **compute cycles** to identify the first dummy atom.

Can we compact the bins to use what they need?

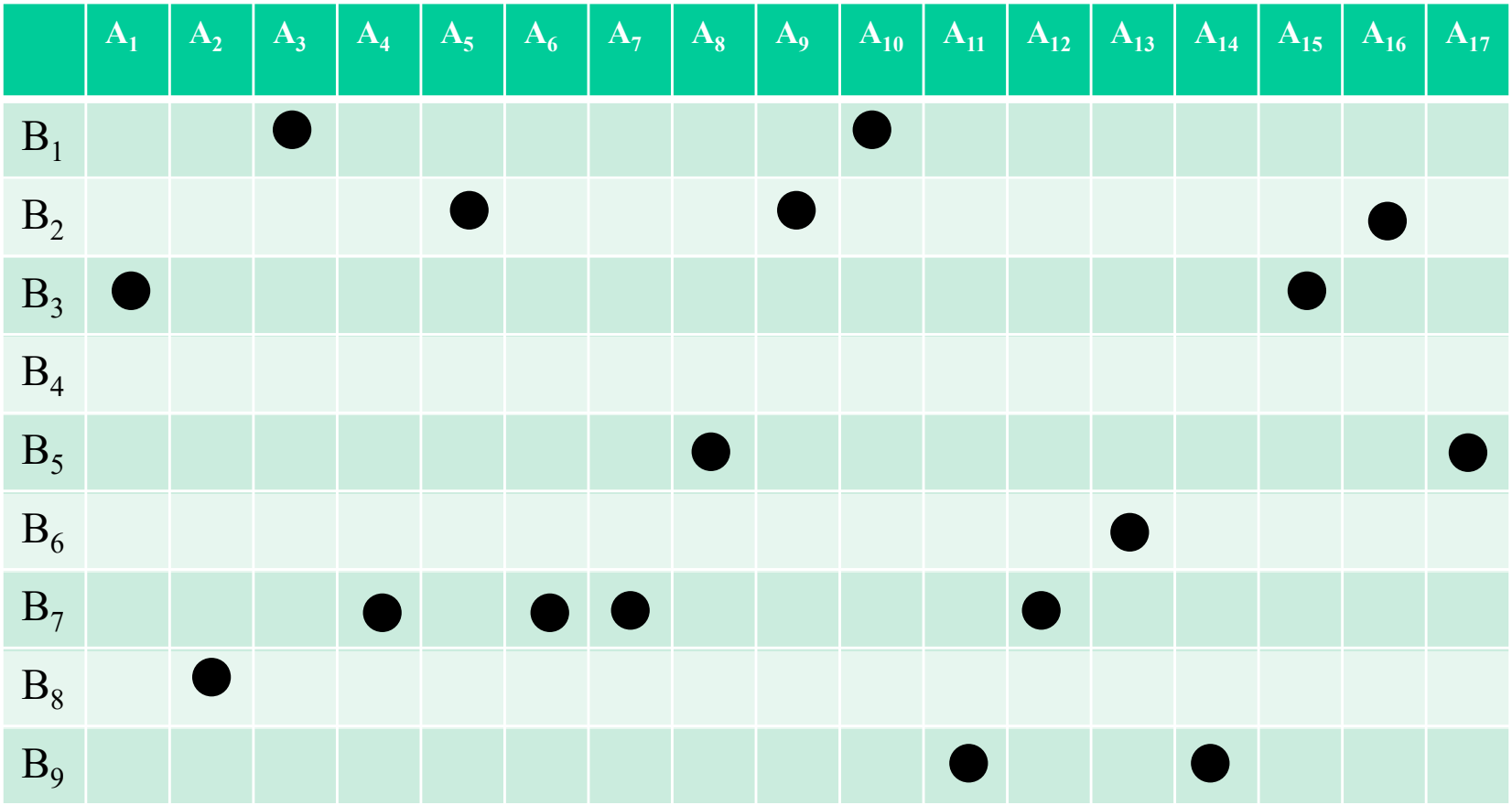
What Does Bin Compaction Mean?

Think about **relationship between atoms and bins**:

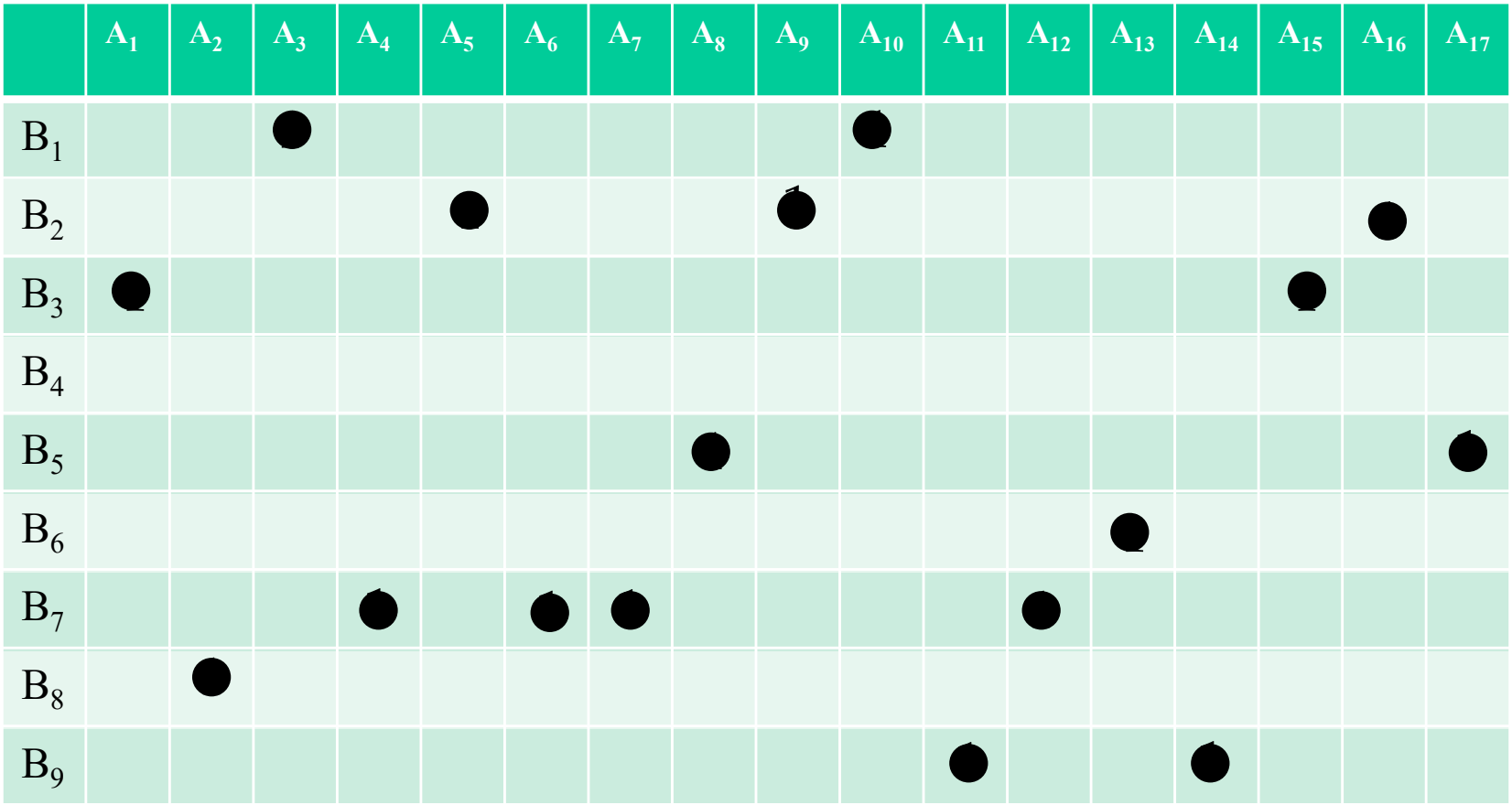
- Every **atom** goes **into a single bin**
- Every **bin contains only a few** atoms (small number compared to whole set).

Remind you of anything?

Look Familiar Now?



What About Now?



Maybe with Some Zeroes?

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}
B_1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
B_2	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0
B_3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
B_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B_5	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
B_6	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
B_7	0	0	0	1	0	1	1	0	0	0	0	1	0	0	0	0	0
B_8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B_9	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0

Represent Bins Using a CSR Format

Looks a bit like a sparse matrix!

Remember sparse matrix formats?

- Rows (bins) have few non-zero elements (atoms).
- **Use** a compressed sparse row (**CSR**) **format**.

Put **4-tuples** (X, Y, Z, charge) **in place of** matrix **values**.

Also,

- atom "order" doesn't matter, so
- **permute columns** instead of indexing them.

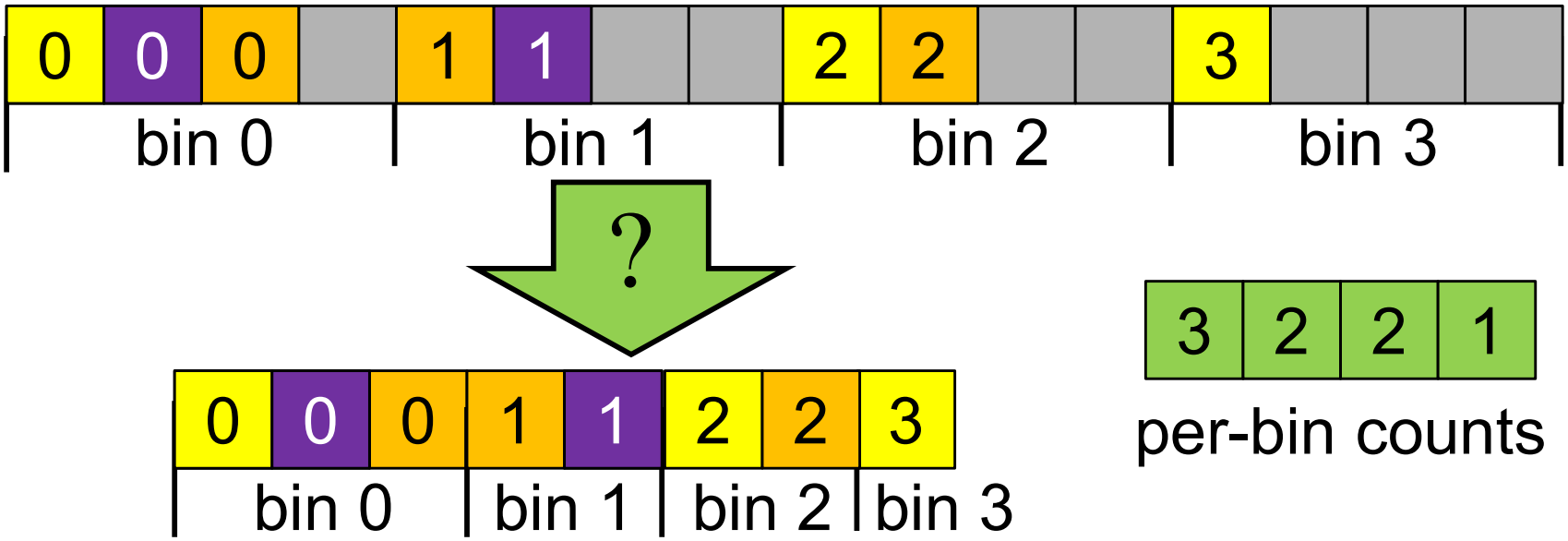
Something Like This…

	A_3	A_{10}	A_5	A_9	A_{16}	A_1	A_{15}	A_8	A_{17}	A_{13}	A_4	A_6	A_7	A_{12}	A_2	A_{11}	A_{14}
B_1	A_3	A_{10}															
B_2			A_5	A_9	A_{16}												
B_3						A_1	A_{15}										
B_4																	
B_5								A_8	A_{17}								
B_6										A_{13}							
B_7											A_4	A_6	A_7	A_{12}			
B_8															A_2		
B_9																A_{11}	A_{14}

Bin Counts from Histogram Give Size of Bins

But we only learned how to compute with CSR!

How can we create a CSR format?



Three-Step Process for Compact Binning

Conceptually, we have the following:

1. Compute histogram
   - of number of atoms in each bin, and
   - save per-atom indices.

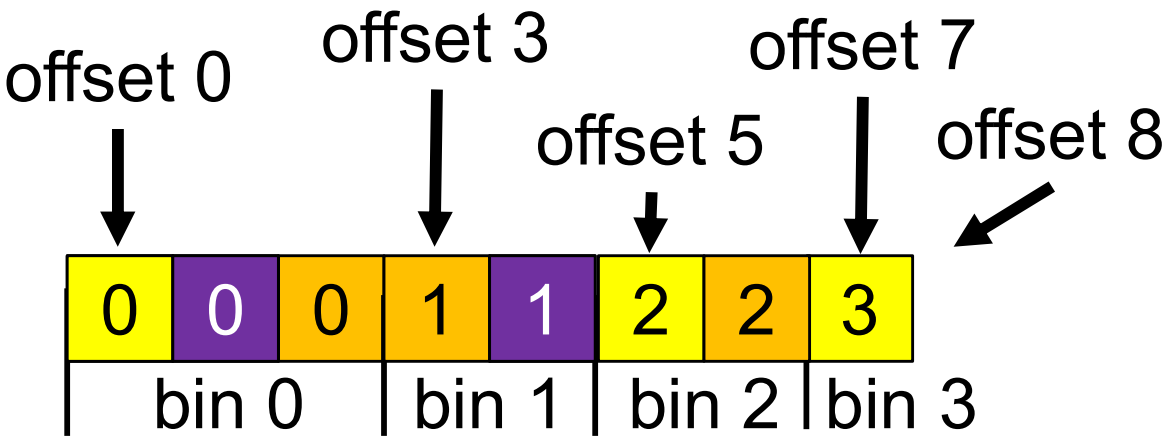
1.5. Figure out where bins should go in memory.

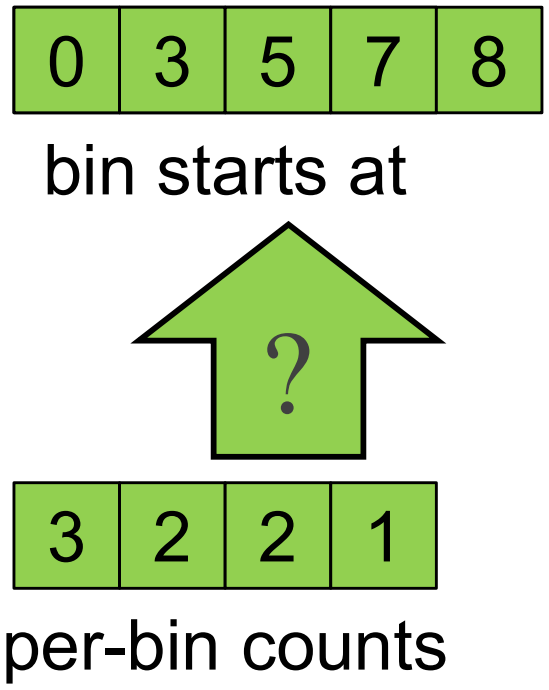
2. Use per-atom indices
   - to copy each atom's data
   - to its assigned location in a bin.

Compute Bin Starts from Bin Counts?

Need to find starting indices for each bin.

Can we compute bin starts from per-bin counts?



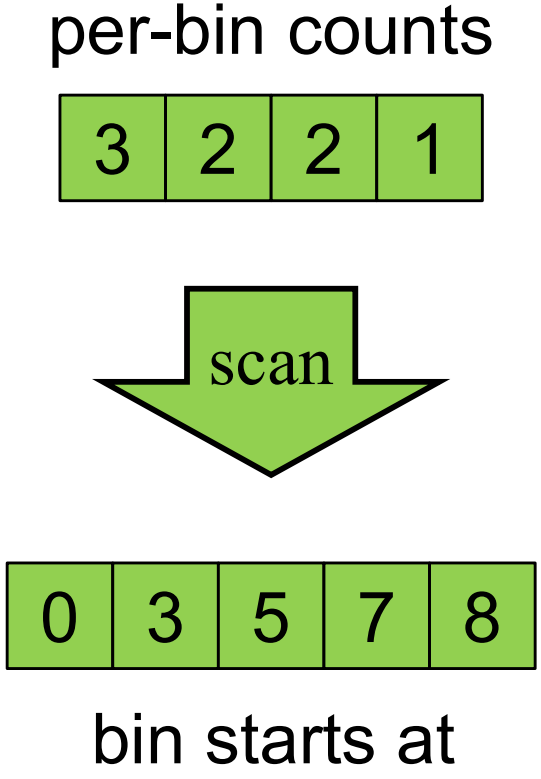


Step 1.5. Determine Start and End of Bins

Use **parallel scan**

- **on** the **bin counts**
- **to generate** an array
- of **starting points**!

Parallel scan is a 408 lab,
as is histogram.



Important Enough to Use in Theory

"… scan operations, also known as prefix computations, can execute in no more time than … parallel memory references … greatly simplify the description of many [parallel] algorithms, and are significantly easier to implement than memory references." —Guy Blelloch, 1989*

*G. Blelloch, "Scans as Primitive Parallel Operations," IEEE Transactions on Computers, 38(11):1526-1538, 1989.
The idea behind scans for computation goes back another 30+ years.

Trying to Bridge Theory and Practice

A generic parallel **algorithm**,

- **in which** parallel threads **access memory arbitrarily**,
- **is** likely to produce an **extremely slow** access pattern.

Scans

- can be implemented **quick**ly **in hardware,** and
- form **a useful alternative** to arbitrary memory accesses.

(His hope was to enable theory
without knowledge of microarchitecture.)

Three-Step Process for Compact Binning

Conceptually, we have the following:

1. Compute histogram
   - of number of atoms in each bin, and
   - save per-atom indices.

1.5. Parallel scan of bin counts.

2. Use per-atom indices
   - to copy each atom's data
   - to its assigned location in a bin.

Sort Atoms into Bins without Conflict

Atoms can then be placed into bins in parallel:

```
for each input element Elem {
  BI ← bin index for Elem
  // (check for overflow omitted)
  idx = bin_start[BI] +
        atom_index[Elem];
  insert atom into bin[idx]
}
```


Three-Step Process for Compact Binning

That's not what is done in Lab 4, however.

Lab 4 uses an unmodified histogram for Step 1,
and does not save per-atom indices,
so Step 2 (sort) requires use of atomics again.

(These parts are all extra credit, also.)

Algorithm for Computing with Compact Bins

How do we process compact bins?

For each (Δy, Δz)

- Load (from constant memory)
  - relative indices of
  - starting/ending bins in X pencil
  - of containing sphere.
- Add each value to thread block's index, then
- look up starting/ending indices in compact atom array.
- Finally, tile access to atoms (8 to 32 atoms per step).

Another Problem?! Yes, Load Imbalance

One last problem: load imbalance!

- With non-uniform atom distributions,
  - **Some bins contain many atoms**, and
  - Many bins contain few atoms.

- Leads to load imbalance:
  - **thread blocks near bins with many** atoms
  - **take much longer** than those far from such bins.

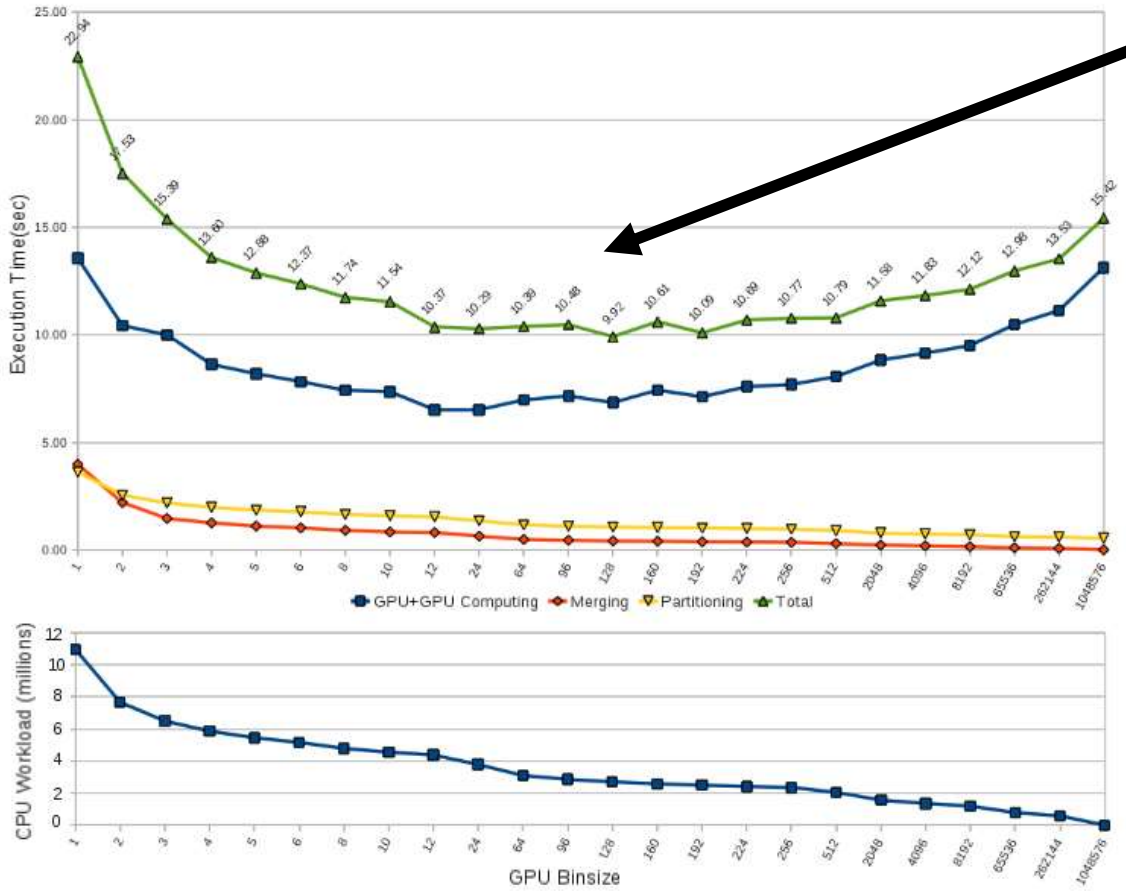
Limit Bin Capacity to Limit Load Imbalance

Solution? Set a limit on bin size!

- **Limit** the **number of atoms** in each bin.
- If bin exceeds limit, **place in overflow bin** instead (as with uniform bin capacity approach).
- **CPU does computation on overflow bin** in parallel.
- CPU merges results.

Limits load imbalance and
leverages GPU-CPU computing power.

Performance Not Too Sensitive to Bin Capacity Limit



green line is total execution time

Performance Thoughts

- Compaction takes time, but **overall computation** is about **30% faster** with compaction.

- Rather than using multiple atomics to implement **overflow detection** on hot bins,
  - **read counter** value **first**, then,
  - **if** its **too large,** go to **overflow**.
  - Otherwise, do atomicAdd or CAS.

Techniques are Reusable in New Contexts

Why does everything turn into a 408 lab?

In reality, there is **rarely a need for new ideas**.

- Instead, **reuse ideas** in new contexts.
- That's **99.9% of engineering and breakthroughs**.

George Polya, a Stanford mathematician,

- even espoused the idea as the **basis of learning**:
- **learn what people have done**
- so that you can **apply those techniques**
- **to new** proofs and **problems**!

ANY QUESTIONS?