ECE 508
Manycore Parallel Algorithms

Lecture 5: Input Binning

Objectives

• to understand how parallelization over outputs (the gather approach) can lead to data scalability problems,
• to examine basic techniques for input binning (also known as spatial sorting), and
• to understand common tradeoffs in input binning.

Scatter to Gather Transformation

Obvious Approach to Gather is Not Scalable

• Recall: inputs often less regular than outputs, and
  – sometimes difficult to identify relevant inputs, or
  – to exclude those inputs that are NOT relevant.
• A naïve option:
  – process all inputs to decide relevance.
  – Execution time scales poorly with data set size,
  – leading to data scalability issues.

Lack of data scalability is a major problem for GPUs: who needs a GPU to solve a small data set?
Not Just for HPC Any More!

• We examine **binning**
  – in context of electrostatic field computation on a GPU.
  – The more sophisticated techniques are from HPC.
  – Basic techniques (first) you may already know.
  – They’re **useful in all contexts**.
• For example: after finals in Dec. 2007, I wrote a racing game in ActionScript 2 (Flash) for my car-fanatic nephews. When the number of objects in the world grew too large, it got slow. The solution? Binning. Problem solved!

Binning Also Invaluable for Big Data

Also (of course!) used for “big data”…

• datacenters,
• social networks,
• content distribution networks, and
• on and on…

Many engineers focus almost entirely on how best to distribute and manage data for scalability.

What Are We Measuring?

Before we dive into the strategy,
• let’s understand the problem that we’re trying to solve.
• We’ll start with some standard terminology from high-performance computing,
• then look at some newer ones.

Speedup Measures the Success of Parallelization

• Let’s start by defining **parallel speedup**
  (usually just called speedup).
• Let’s say that
  – when I run my program in parallel
  – it finishes \( X \) times faster
  – than when I run it sequentially.
• Specifically,
  – \( X = \frac{T(\text{sequential})}{T(\text{parallel})} \), and
  – \( X \) is the speedup of my parallel code.
Scalability Measures Effect of Parallel Overheads

- What is **scalability**?
  - **For how many processors is speedup linear**?

At some P, with fixed problem size, speedup will flatten out.

---

Speedup Measures Improvement for an Input Set

- **Speedup assumes a fixed problem size**.
  - For many applications, that’s reasonable.
  - Users care about their input sets, not about hypothetical inputs.

- But that’s not always the best assumption, and other versions have been suggested.*


---

Learn How to Measure Success

Lots of tricks can be used to make results look good!

- Most have been known for 30+ years.*
- As **graduate students**, you should
  - **understand** the alternative approaches,
  - **choose** the right one, and
  - be able to **defend your choice**.
- The approach we’re about to describe, for example, is listed as one of the tricks…


---

For Other Situations, We Need Different Metrics

**Sometimes we care about throughput:**
- frames per second for video / game quality,
- transactions per second for databases, or
- user operations per second for datacenters.

**GPUs are throughput-oriented.** **Data scalability** means
- **throughput independent of data size**.
- Not a new idea*, but important in our context.

Linear Complexity Required for Data Scalability

**Why is data scalability important?**

- **Large data sets** a major motivation for parallel computing.
- Even $O(N \log N)$ can be a problem.
  - With millions of data points, $\log N = 20$.
  - With billions, $\log N = 30$.
- Want constant throughput solution—that’s data scalability.

Algorithmic complexity above linear is not data scalable.

---

**Complexity vs. Data Scalability**

0 50 100 150 200 250 300 350 400 450 500

0 50 100 150 200

Data Size

Computation Time

- Quadratic (32-way parallel)
- $N \log N$ (4-way parallel)
- Linear

**DCS Algorithm: Accurate and Highly Parallel**

**Remember DCS?**

- At each grid point, sum the electrostatic potential from all atoms.
- All threads read all inputs.
- Highly accurate and data-parallel!

**DCS Algorithm Complexity?**

- number of grid points $\times$ number of atoms
- both proportional to volume $V$
- compute time proportional to square of volume $V$!

*Poor data scalability!*
DirectSum (DCS): Accurate, but Poor Data Scalability

- GPU DCS slower than CPU with cutoff.
- Caveat: old GPU, so your code should do better in an absolute sense.

Ignore Atoms Beyond a Cutoff Distance

- For biomolecules in solvents, and many intermolecular forces, we can ignore atoms beyond a cutoff distance $C$, with $C \approx 8\text{Å}–12\text{Å}$.
- Compute long-range potential separately (if desired) using, for example, Multi-Level Fast Multipole Algorithms.

With Cutoff, Work Scales Linearly with Volume

- Number of atoms:
  - within cutoff distance
  - roughly constant (uniform atom density)
  - 200 to 700 atoms within $8\text{Å}–12\text{Å}$ cutoff sphere for typical biomolecular solvents.
- Amount of work scales linearly with volume!

Example of Cut-off Summation

Electrostatic potential $V$ at position $r$ given by:

$$V(\vec{r}; \vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) = \sum_{i=1}^{N} \frac{q_i}{4\pi\varepsilon_0 |\vec{r} - \vec{r}_i|} s(|\vec{r} - \vec{r}_i|)$$

1 in proper units. Permittivity of free space for engineers.
Example of Cut-off Summation

Electrostatic potential $V$ at position $r$ given by:

$$V(\vec{r}, \vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) = \sum_{i=1}^{N} \frac{q_i}{4\pi\varepsilon_0 |\vec{r} - \vec{r}_i|} s(|\vec{r} - \vec{r}_i|)$$

$$s(r) = \begin{cases} 
(1 - r^2/r_c^2)^2, & \text{if } r < r_c, \\
0, & \text{otherwise}
\end{cases}$$

Smoothing function to ensure conservation of energy in dynamics.

Implementation Challenge: Don’t Look at Atoms

For each grid point, identify the atoms that need to be summed.

- Seems simple enough. So …
  - compute distance for each atom, and
  - discard those beyond the cutoff?
- No. That’s quadratic.
  
  A thread must not examine atoms outside the range of its grid point(s) at all!

Use Binning/Spatial Sorting to Simplify Selection

The solution?

- **Group data into bins.**
- Each bin represents a property for data in the bin.
- Bins coarsens data to simplify selection.

Variants: uniform spatial bins (next), variable-size bins, KD Trees, …

Uniform Binning for Cut-Off Potential

Divide simulation volume into uniform, non-overlapping cubes.

- Bins (cubes) represent location of atoms.
- Each cube has a unique index for parallel access.

Map each atom to a cube based on spatial location.
Spatial Sorting Gives Linear Complexity

- Atoms placed in bins
  - before electrostatic computation.
  - Each bin can hold any number of atoms.

- Cutoff potential for a grid point
  - only considers bins
  - partially/fully within cutoff sphere.

Result: linear complexity.

Binning Reorganizes Data to Improve Performance

**Binning means data reorganization.**

This example is
- the first time that
- we’re moving data around
- to improve performance
- (not counting transpose of \(M\) in GEMM).

As mentioned earlier, the most important part of one’s job in many companies is to figure out how to organize data.

Each Bin Holds a Fixed Number of Atoms

Let’s start simple.

- **CPU maps atoms to bins.**
- Each bin holds a fixed number of atoms, the bin capacity.
- Atoms that don’t fit
  - placed into a single (arbitrary-size) overflow bin, and
  - processed by CPU (in parallel with GPU).
- Could instead chain overflow into additional bins
  (linked rather than spatially indexed).

Bins are Like Classrooms…

A bin is like a classroom:
- Each room has a fixed size.
- The fire marshal decides how many can fit.
- University not allowed to enroll more students
  (cannot oversubscribe) for the room,
- even though we know that
  - not all students
  - will show up every day!

The overflow bin? An arbitrarily-sized online section!
(A different one per class, of course.)
Balance Capacity Based on Bin Size/Atom Distribution

**Capacity** of atom bins needs to be balanced:
- too large → many dummy atoms in bins, but
- too small → too many atoms in overflow bin.
- **Choose** capacity to cover >95% of atoms.

Expected **number of atoms** per bin depends on
- distribution of atoms, and
- **bin size, B**: dimension of bins (cubes) in simulation volume.

Uniform Bin Size Further Simplifies Implementation

Again, for simplicity,
- we use **constant B**,  
- making it easy to translate spatial coordinates to bin coordinates.
Together with constant bin capacity, also  
- allows array implementation  
- and (direct) use of relative bin offsets  
(discussed later).

Bin Size Provides Another Tuning Parameter

**Larger bins** cover more space, and  
- contain larger (expected) numbers of atoms,  
- smoothing the statistics for choosing bin capacity.  
- Larger bins also contain more grid points, and are thus connected to thread block size.

Threads still check the cutoff distance  
- screen away atoms outside  
- However, if B is too large, threads examine too many atoms out of range.  
- Making the code inefficient.

Each Bin Computed by a Thread Block

Parallelization strategy:  
- four grid points per thread,  
- and thread block covers grid points in one bin.  
Ask: what bins are  
- within the cutoff  
- for one or more grid points in a given bin?  
(Bring each other bin into shared memory.)
**Simplify the Problem**

First, let’s simplify.

The function is homogeneous based on bin coordinates: solve for one bin, and we know answer for all bins.

The function is symmetric:
- +/- X, Y, and Z, or
- swap any pair of X, Y, Z.

**Compute Loops Bounds in Each Dimension**

Let Δx be X difference in bin coordinates (same for Δy, Δz).
Shape is convex and symmetric, so just identify maximum magnitudes.
Loop order will be Z, Y, X:
1. **Compute Z loop bound** (maximum Δz magnitude), then
2. For each Δz, compute Y bound,
3. For each Δy, Δz, compute X bound.
   (Can use symmetry to simplify further.)

**Find Minimum Distance to an Atom**

Let’s find the Z loop bound.
- Δz ≥ 0 case: measure from grid point with max Z coordinate (on right).
- **Compute minimum dz** to atom in another bin.
- We have
  - dz ≥ B(Δz – 1) + G/2
  - and dz ≥ 0 (for Δz = 0)
- minimum dz is then
  \[
  dz_{\text{min}} = \max (0, B(\Delta z - 1) + G/2)
  \]

**Finding an Offset Limit for Loops**

- From \(dz_{\text{min}} = \max (0, B(\Delta z - 1) + G/2)\), we can compute the maximum value of \(\Delta z\) (when \(\Delta x = \Delta y = 0\)).
- Given cutoff C, we require \(dx^2 + dy^2 + dz^2 < C^2\), which simplifies to \(dz^2 < C^2\).
- Substituting and solving for \(\Delta z\) (ignoring the \(\Delta z = 0\) case) gives \(\Delta z < (C - G/2) / B + 1\), for which the maximum is
  \[
  \Delta z_{\text{max}} = \left[ \frac{C - G/2}{B} \right].^*
  \]
  *Technically, we should also bound below by 0, but not necessary for reasonable values of B, C, and G.
## One Option: Use Cube Containing all Grid Points’ Spheres

**One option** is to stop here.
- By symmetry, $\Delta z_{\text{max}}$ is the maximum bin offset magnitude.
- Pass as kernel arg, or compute in threads.
- Resulting **cube of bins** contains all atoms that any given bin need consider.

### Finding an Offset Limit for Loops

Or we could **compute more carefully**.

We know $\Delta y_{\text{max}} = \Delta z_{\text{max}}$ when $\Delta z = 0$, so

- compute $Y$ bound for $\Delta z > 0$ and $\Delta x = 0$.
- $\Delta z$ gives $dz_{\text{min}} = B(\Delta z - 1) + G/2$, so
  - $dx^2 + dy^2 + dz^2 < C^2$ simplifies to
  - $dy^2 + [B(\Delta z - 1) + G/2]^2 < C^2$.
- Solving then gives

$$
\Delta y_{\text{max}} = \sqrt{\left(\frac{c^2 - [B(\Delta z - 1) + G/2]^2}{B}\right)^{1/2} - G/2}
$$

## Second Option: Compute All Bin Offset Bounds

**Solving** in this manner **approximates a containing sphere** more exactly, as shown to the right.

Circles are the cutoffs for the corner grid points with $C = 2.25B$ in the $\Delta z = 0$ plane.

### Containing Sphere Complicated But Efficient

**Containing sphere** method

- based on computed bin offset bounds
- **more accurate**, but requires **complicated** computation.

Fortunately, can **precompute** once on **CPU** (or pre-compile),
- **linearize offsets** once, and
- **store in constant memory**.

Threads then just walk through array of bin offsets and use them to find atoms.
Strategy with Containing Sphere (or Cube)

- Bins stored as list of offsets; list has constant length based on B, C, G.
- Threads in block scan same bins (read same addresses).
- Some divergence:
  - some bins not needed by some grid points, and
  - some atoms within a bin inside of some grid points’ cutoffs, outside of others.
- Threads must still ignore atoms outside of cutoff.

Complexity is Linear for Good Atom Distributions

- complexity of the algorithm: \( O(MN') \)
  - \( M \): number of grid points
  - \( N' \): number of atoms in neighborhood offset list
- In general,
  - \( N' \) is small compared to number of atoms
  - works well if distribution of atoms is uniform (otherwise, can have load imbalance or large overflow bin).

Typical Parameter Values for Grid, Bins, and Thread Blocks

For forces of interest and typical atomic spacing, the following parameters are typical:
- lattice spacing \( G = 0.5 \) Å
- bin dimension \( B = 4 \) Å

Each thread block computes a \((4\,\text{Å})^3\) cube of the potential map:
- 8x8x8 potential map grid points
- using 128 threads per block (4 points/thread).
Why So Inefficient?

Did you expect higher efficiency?
As stated earlier, the numbers on previous slide account for
1. bins outside of grid point sphere (greyed out) and
2. atoms outside of grid point sphere (white circle).
More subtly, remember that many bins are partially empty, but are read into shared memory anyway.

Should Still be Compute-Bound!

Now think in terms of reuse.
• Each bin used for $8 \times 8 \times 8$ grid points: reuse of 512×.
• 36.4% of atoms are useful ($B=4$, $C=12$ Å, sphere).
• Say bins are half full.
0.364×0.5×512 = 93× reuse! O(10) FLOPs/atom.
Plenty of computation!

Design Space Exploration for GPU Applications

You may explore
• other thread block / coarsening organizations.
• Output tiles/atom bins should be cubic to minimize inefficiency.
Tradeoffs form a design space for optimization.
• Early GPUs produced difficult spaces to search,*
• but hardware changes (such as caches) and
• better understanding (such as this class) have
• made the process easier.


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

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

// 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)
```
Optimize Both Memory and Compute Behavior

High memory throughput is essential.
- Group threads together for locality (adjacent points have maximally overlapping spheres).
- Fetch bins of data into shared memory:
  - bins and atoms organized for efficient fetching.
  - and enables reuse, as discussed.

Done a good job? Still need instruction-level optimization for compute-bound kernels!

Same Approach Used in Clusters

Optimizations also enable supercomputing with
- many CPUs and GPUs and
- distributed memory.

Spatial sorting (binning) enables
- distribution across memories, and
- proactive transfer of bins to grid points
- (owner computes, remember?).

Tiling Atom Data

Be careful about exposing latency!

Using shared memory reduces global memory bandwidth:
- thread block collectively loads bin into shared memory,
- synchronize, then
- threads scan atoms in shared memory (and sync. again).
- Lots of reuse!

But loading a bin can take 1,000+ cycles!
Remember to Allow for Multiple Blocks/SM

We have four warps per block, but they hit a barrier!
• **Need several thread blocks** to fully utilize the SM.
• That’s why we don’t want to process
too many points / too many bins at once.
**Occupancy matters!**
• Many CUDA programmers forget that fact.
• Later, we’ll see how to use the NVIDIA tools
to see what’s going on.

Bin Capacity of 8 is a Good Choice

With our example parameters, bins contain $(4 \text{ Å})^3 = 64 \text{ Å}^3$.
• Many interesting systems have
  – nearly uniform density of atoms:
  – just under 1 atom per 10 Å$^3$.
• For water test systems, bins average 5.35 atoms.
**Bin capacity of 8** is a good compromise:
• some bins empty, and
• some bins overfull, but
• most bins mostly full.

Non-Uniform Atom Distributions Require More Complexity

**Average isn’t everything: distribution matters!**
For highly-variable bin capacity,
• we need tree structures, such as octrees.
• Coordinate to bin mapping and intersection
testing become more complex.
  – We will explore those problems later.
  – And, in Lab 4, you’ll implement an intermediate step.

Bin Capacity 8 Also Good for Memory Performance

8 atoms/bin also **good for memory performance**:
• atom is a 4-tuple of floats, so **bin is 128B**, and
• bins are stored **in** a (logical) **3D array**.
For each bin,
• **32 threads** in each block **load a float**
• and store into **shared memory**,
• **full memory bandwidth** with coalesced loads,
• then bin is processed by all threads in block.
Loading More Possible, But May Expose Latency

With 128 threads per block,
- one could load 4 bins instead of 1
- (each thread loads one float).
- Requires 4× the shared memory,
- but only 512B / block.

Bigger issue:
- 4 bins take longer to load, and
- reducing occupancy may expose latency.

Using Both CPU and GPU is Faster

In practice, 2.6% of atoms exceed bin capacity.
- These atoms go into overflow bin
- and are processed using optimized CPU code,
- requiring ~66% of GPU compute time, then
- CPU performs final integration of grid data.

Using both GPU and CPU for computation increases performance overall.

Coarsen in the Y Dimension

What about instruction-level optimization?
First, apply thread coarsening
- to reduce redundant computation and
- to reuse components of distance calculations.

In which dimension?
Y (or Z) – we’ll do 4-way in Y

Avoid Coarsening in X When Locality Matters

Why not X?
Want thread accesses to potential grid to coalesce.

But we coarsened in X for DCS Gather!
There locality didn’t matter, so
- We coarsened with points at least a half-warp apart
- And still coalesced accesses.
- Generally, don’t coarsen in smallest linearization dimension of data unless locality doesn’t matter.
Use the Cylinder Test to Discard Atoms Quickly

**Note:** Y direction is now out of the slide.
(So a thread’s grid points are one point in the figure.)

The **cylinder test**:
An atom outside of the circle is also outside of all of the thread’s spheres.

---

Cylinder Test Divergence Outweighed By Benefits

**What about divergence?**
- Upper two rows of points form the first warp.
- **Divergence** happens where the circles do not overlap.
  - Inside: no divergence.
  - Outside: no divergence.

If you’re going to fail, fail fast!

---

Summary of Code Optimizations for Computation

**Summary of optimizations:**
- threads **compute** four grid points in Y,
- reuse X and Z distance components,
- check combined X and Z components against cutoff *(cylinder test)*, and
- **coalesce stores** for final results.

And…
- **exit loop when first “dummy” atom** found in bin (charge of 0).

---

(Simplified) GPU Kernel Inner Loop

```c
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 */
}
```
Cutoff Summation Runtime

- GPU cutoff with CPU overlap: 12× to 21× faster than CPU core

Cutoff Summation Speedup

- Cutoff summation alone 9× to 13× faster than CPU

Summary of Uniform Binning

- Use of bins requires more sophisticated kernel code to traverse list of bins.
- Bins provide
  - modest work efficiency, and
  - serve as tiles for locality and reuse.
- CPU processes overflow atoms—our first use of joint CPU-GPU heterogeneous computing.
- Trend: spatial tree structures increasingly attractive
  - as new GPU capabilities such as dynamic parallelism mature.
  - We’ll examine in later lectures.

ANY QUESTIONS?