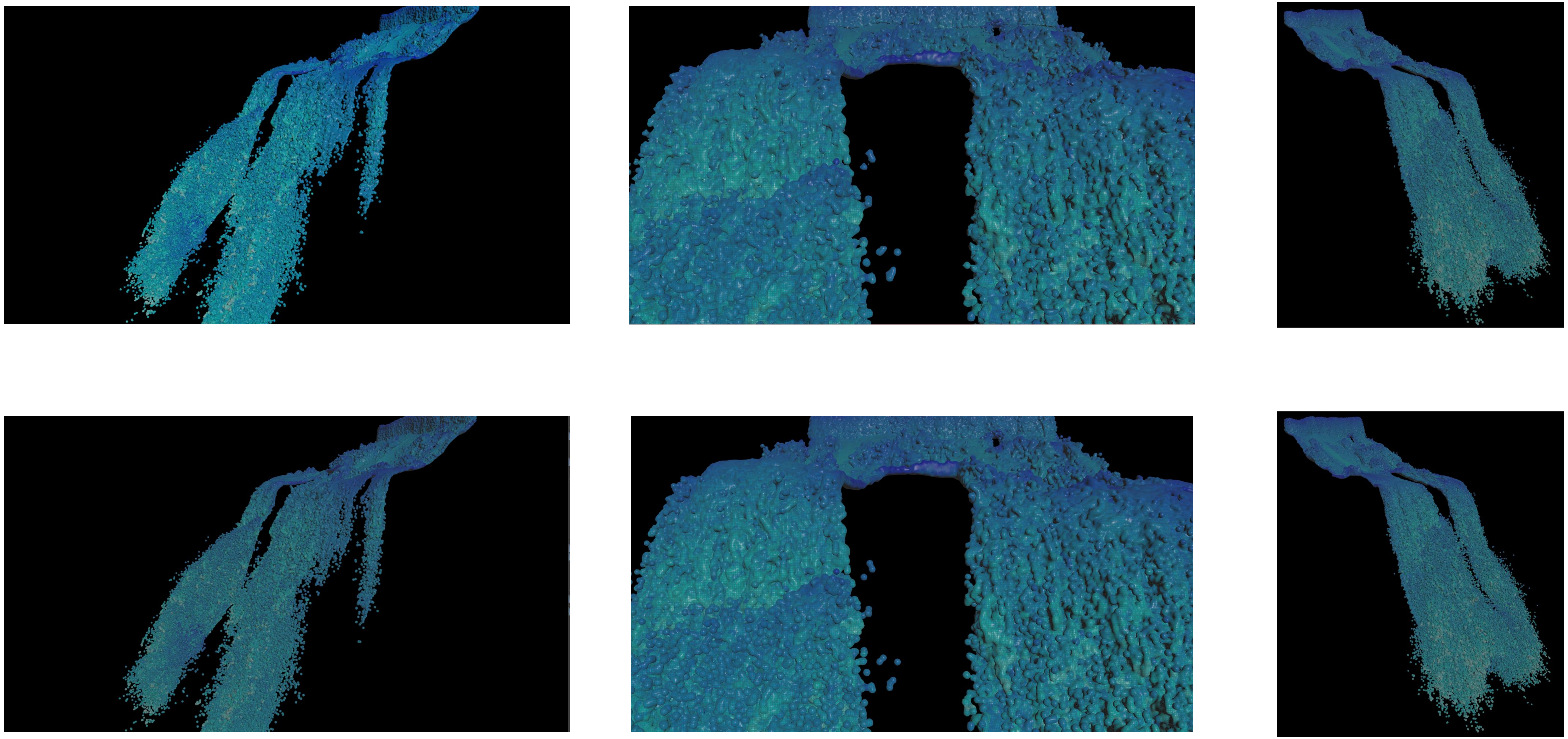
Building NanoVDBs on the GPU

Greg Klar, Ken Museth

Use case: Particle Rasterization



Top: NanoVDB rasterizer; Bottom: OpenVDB rasterizer

Building a NanoVDB from points

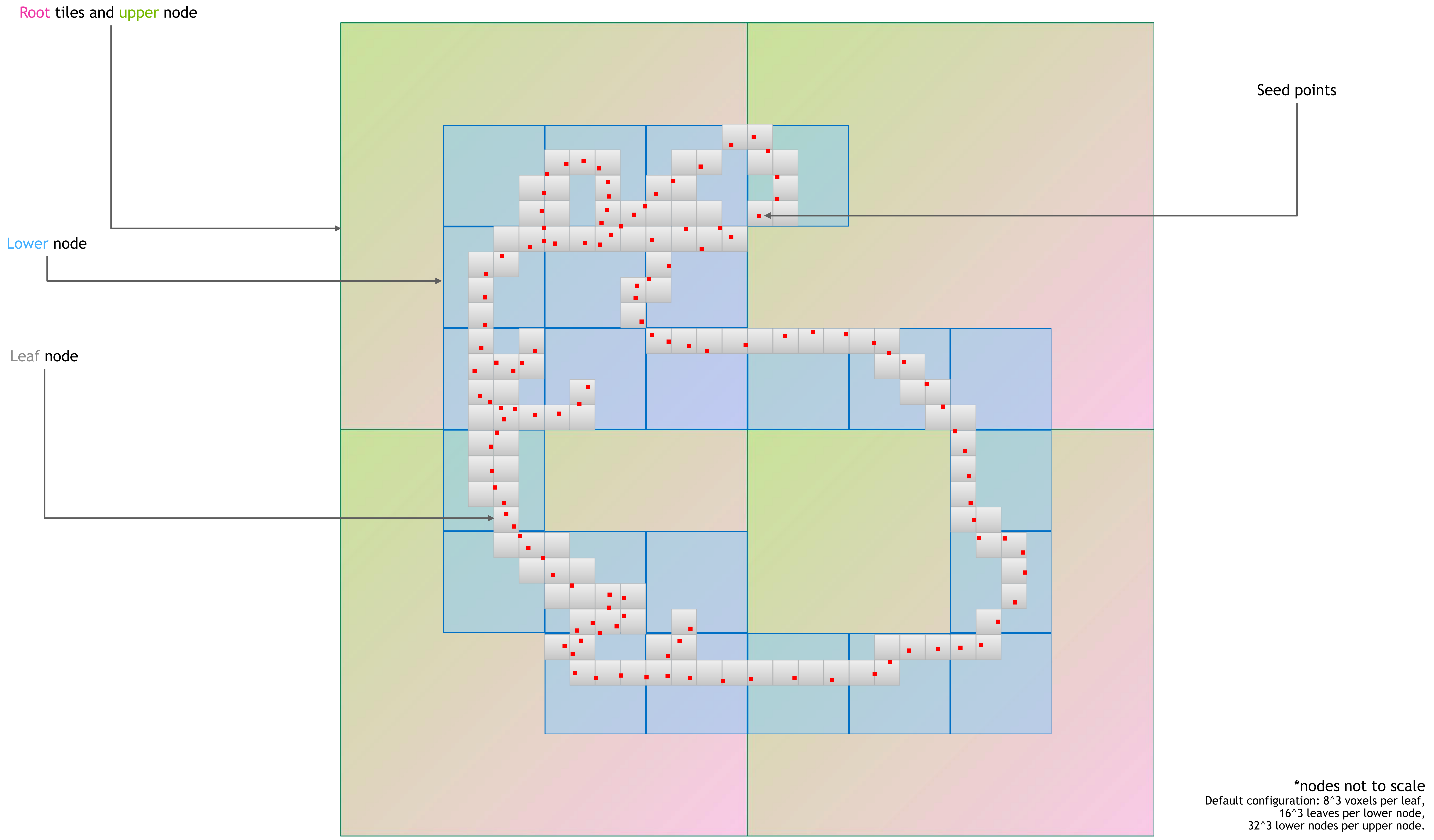
- Snapshot of the source under QR code
- Supports building of regular grids, point grids, index grids
- Related use cases:
  - Point rasterization
  - Point-to-grid transfers

NanoVDB Principles

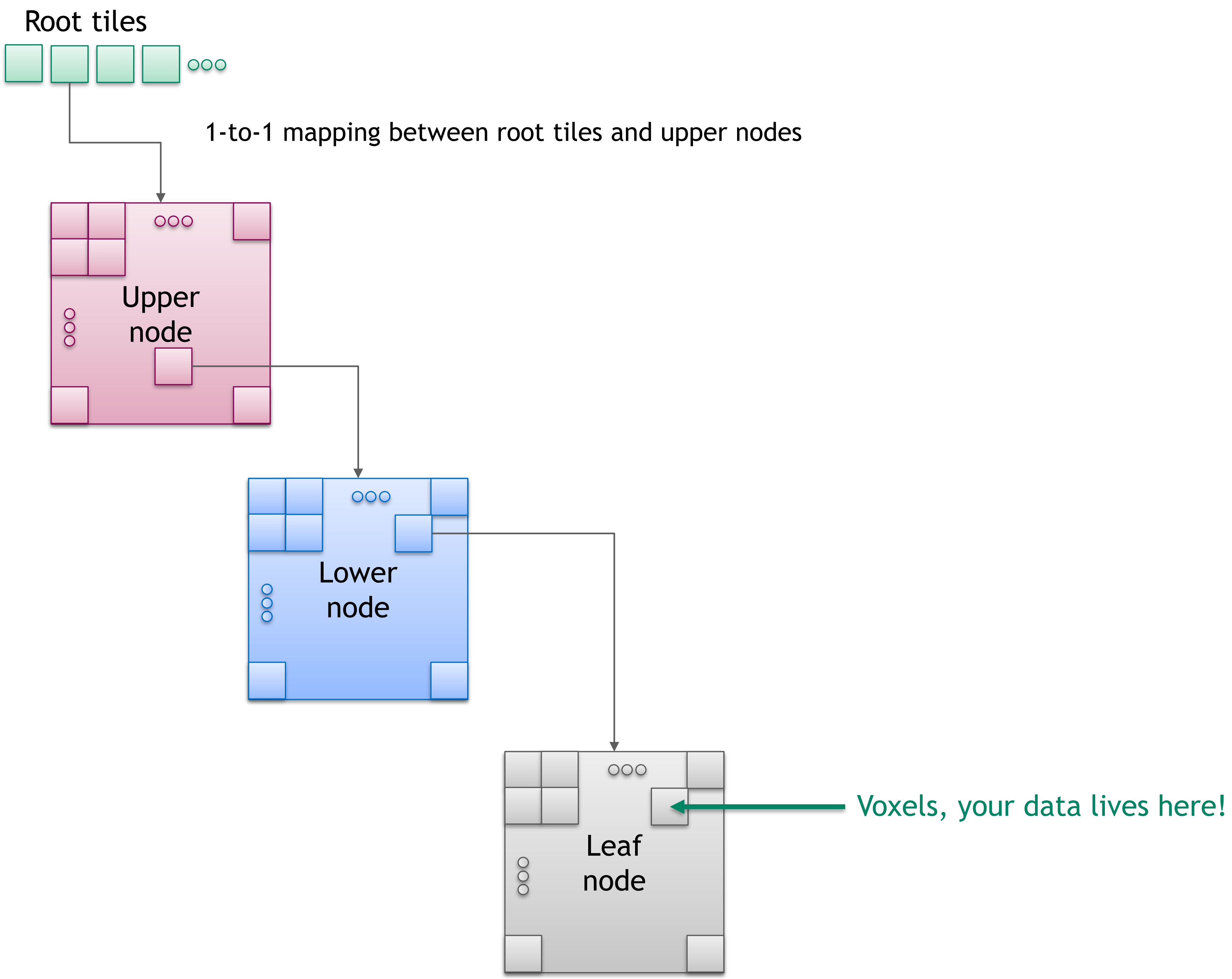
- Pointerless: uses relative offsets in memory
- Very versatile across architectures
- Not well suited for incremental building
- Consequence: Need to know the memory footprint of the grid first!

NanoVDB Building Steps

- Allocate memory
- Build tree and populate values


Root tiles and upper node
Lower node
Leaf node
Seed points
*nodes not to scale
Default configuration: 8^3 voxels per leaf,
16^3 leaves per lower node,
32^3 lower nodes per upper node.

NanoVDB Nodes



*nodes not to scale

How Much Memory to Allocate?

NanoVDB footprint

```
total_bytes =
        sizeof(Grid) +
        sizeof(Tree) +
        sizeof(Tree::RootType) +
        sizeof(Tree::RootType::Tile) * upper_node_count +
        sizeof(Tree::Node2) * upper_node_count +
        sizeof(Tree::Node1) * lower_node_count +
        sizeof(Tree::Node0) * leaf_node_count +
        blind_data;
```

Same in this use case

This is what we need to know!

NanoVDB Building Steps

Corrected

- Count nodes
- Allocate memory
- Build tree and populate values

COUNTING NODES

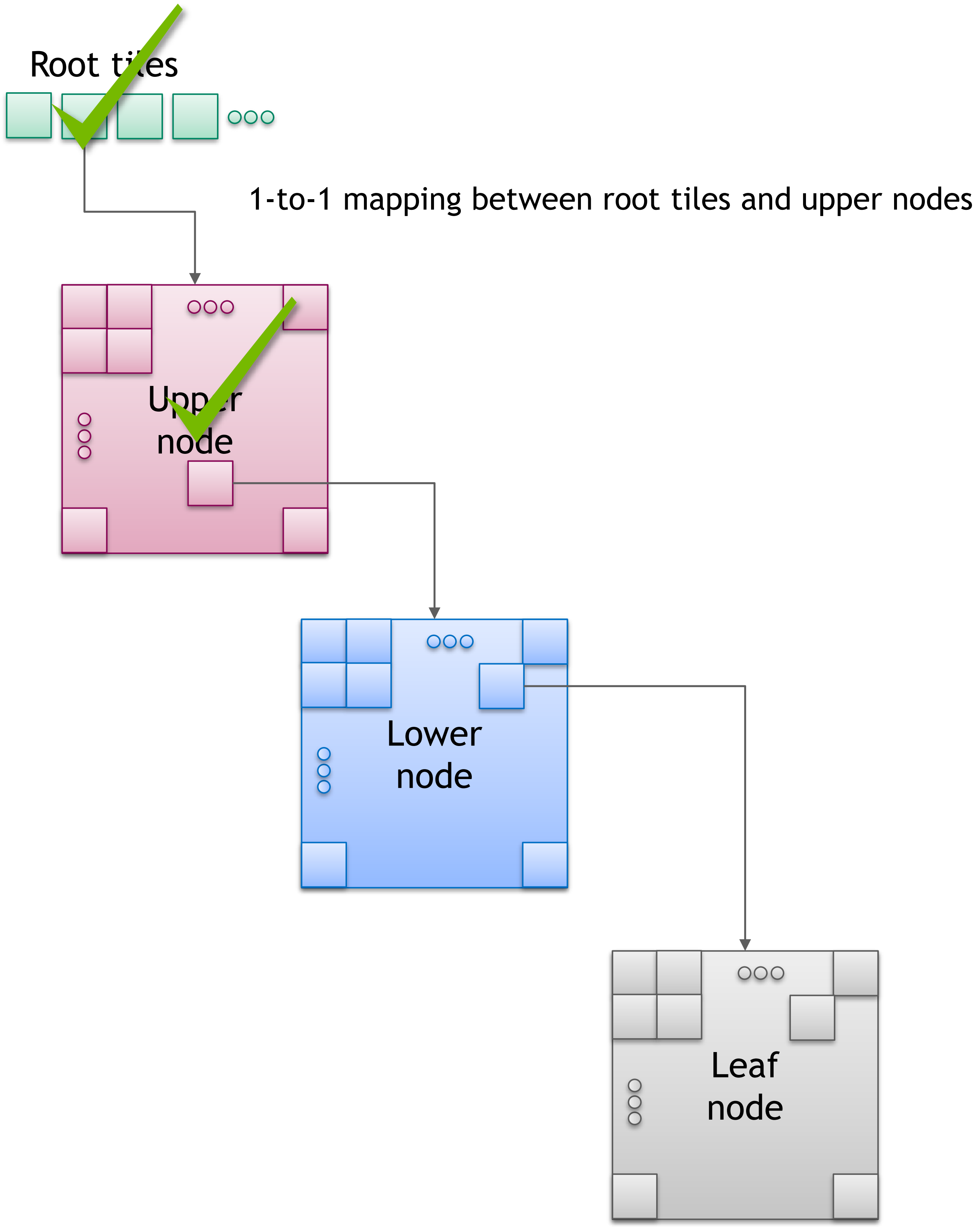
- INTERMISSION -

Binning points on the GPU

- Sort + RLE + PrefixSum = binning
- RadixSort:
  - Sort based on a key the defines the binning
$\longrightarrow$ elements in the same bin will be consecutive
- Run Length Encode
  - $\longrightarrow$ number of elements per bin and the number of bins
- Exclusive Sum (aka *PrefixSum* aka *Scan*)
  - $\longrightarrow$ indices to the start of each bin in the sorted array
- All these are available in CUB!

COUNTING ROOT TILES

- Binning the particle IDs by their Root Key
  - Root keys are available from `nanovdb::RootData::CoordToKey`
- Steps:
  - Generate (*root key, point ID)* pairs for each point based on their index-space location
  - Radix Sort pairs base on *root key*
  - Run Length Encode, outputs:



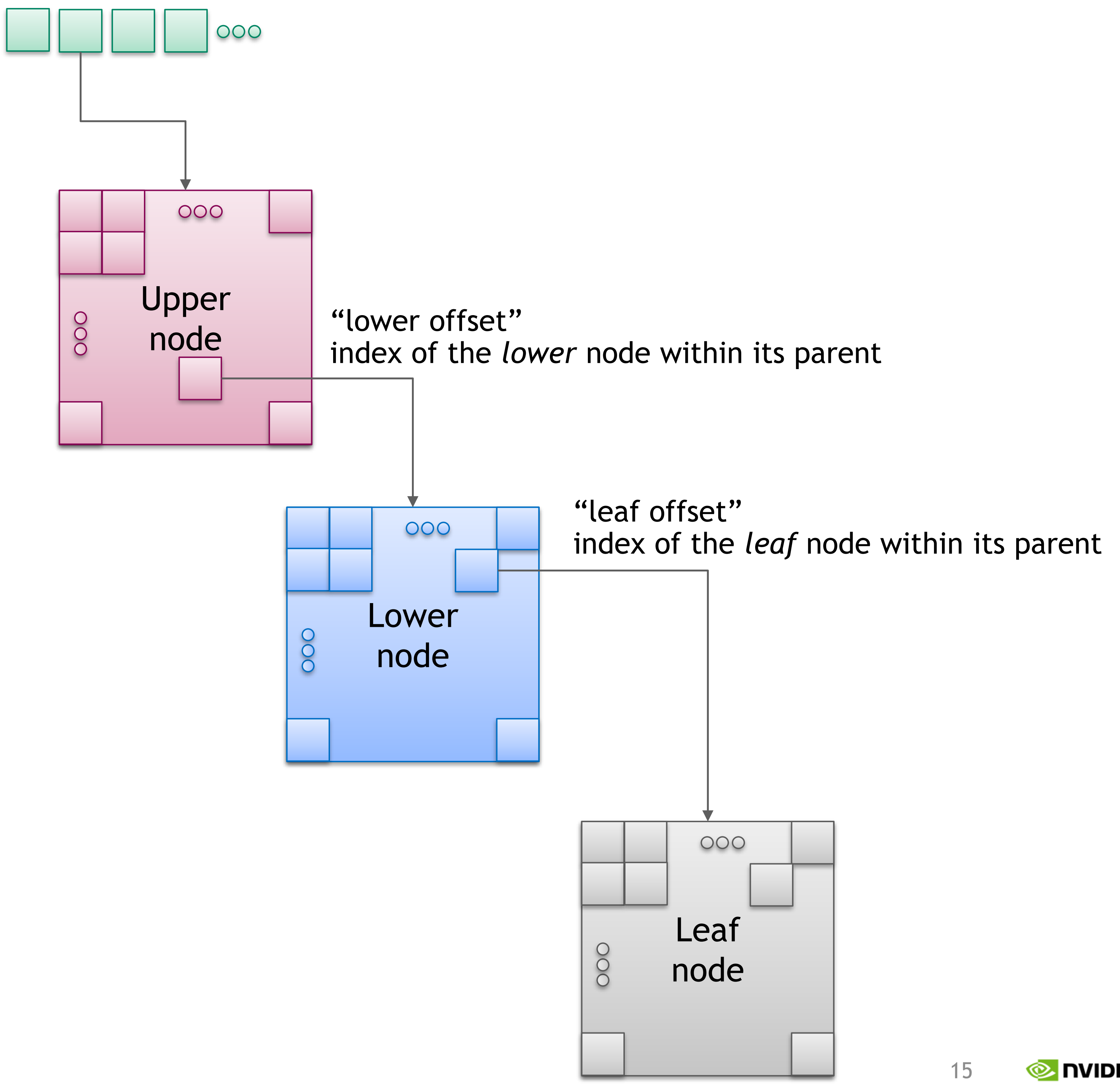
*nodes not to scale

INDEXING WITHIN A TILE

- New 64 bits *voxel key* for each point:

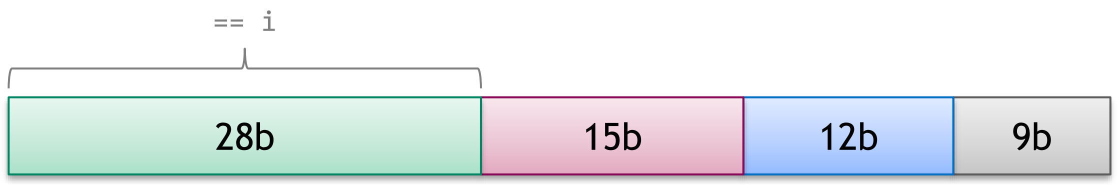
28b	15b	12b	9b

  - 9 bits for *voxel offset*
  - 12 bits for *leaf offset*
  - 15 bits for *lower offset*
  - 28 bits for *tile ID*
    Not the same as tile key!
    This is the running index from `0..tile_count-1`



COUNTING ACTIVE VOXELS

- Plan: bin points to voxels → number of unique bins == number of active voxels
- For each root tile *i*
  - Compute voxel key for each point in the tile



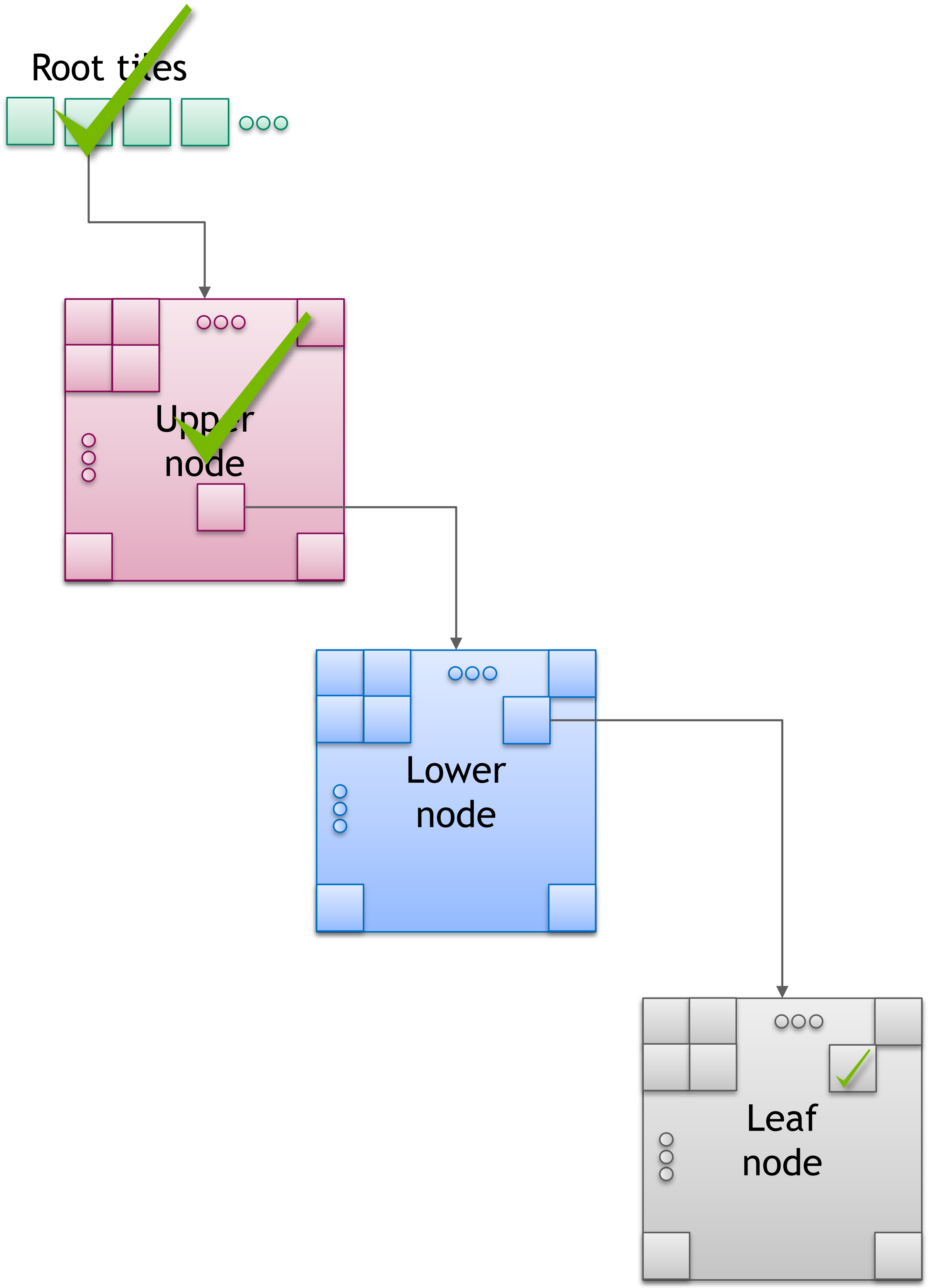
  - Sort voxel keys within the tile to get d_keys d_indx
- Run Length Encode d_keys , outputs:

pointsPerVoxel voxelCount

- Exclusive Sum: offset to look up points based on voxel

pointsPerLeafPrefix

NVIDIA.



*nodes not to scale

COUNTING LEAF NODES

- Note
  - Voxel keys are already sorted in `d_keys`
  - Recall voxel keys are *tile ID, lower offset, leaf offset, ~~voxel offset~~*
  - Shift them right by 9 bits → leaf keys

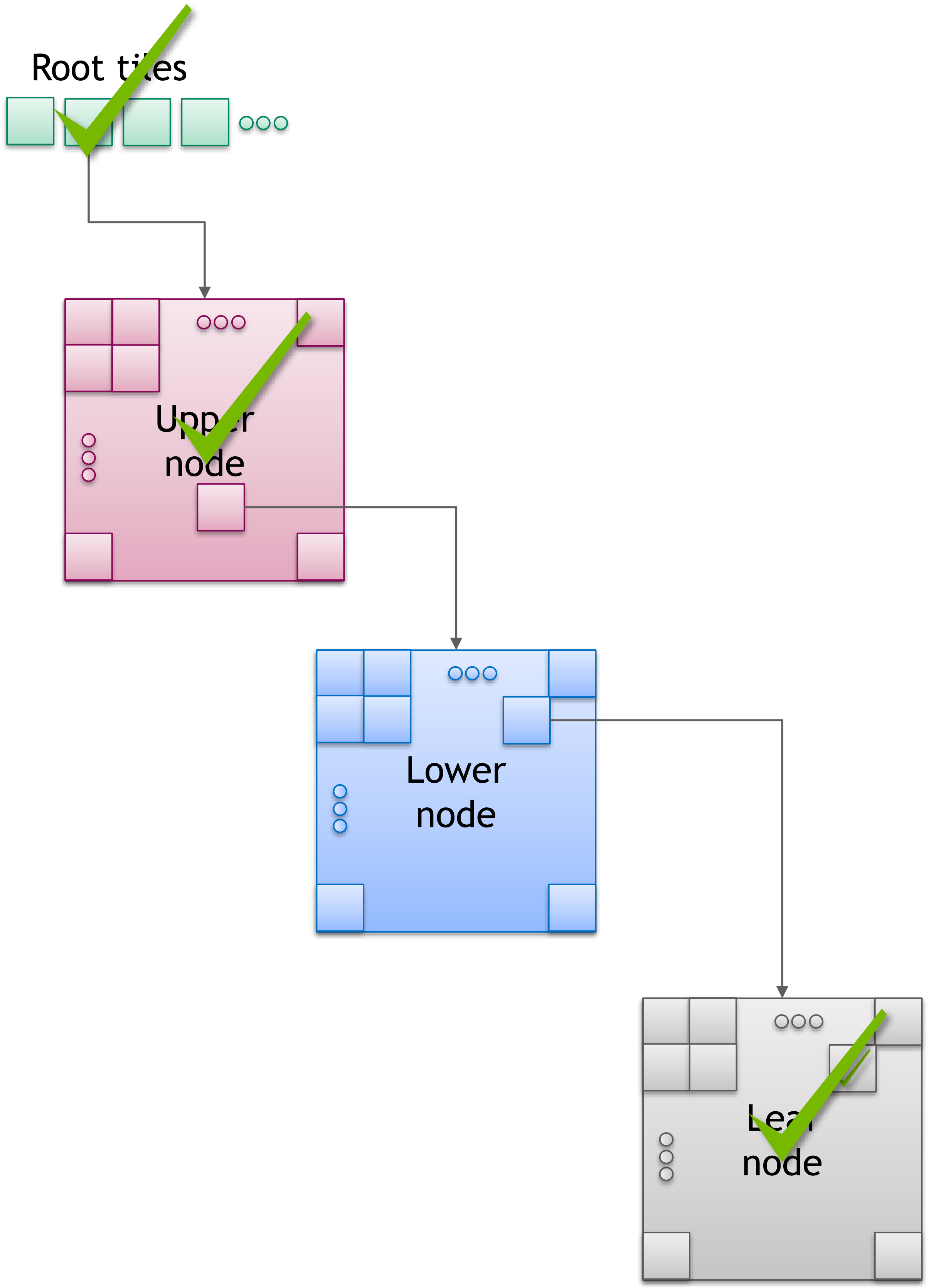
28b	15b	12b	9b

- Run Length Encode `d_keys` with a *Right Shift 9 Bits Iterator*, outputs:

  `d_leaf_keys` `pointsPerLeaf` `leaf node count`

- Exclusive Sum on `pointsPerLeaf` : offset to look up points based on leaf node,

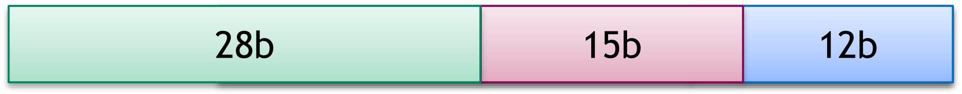
  `pointsPerLeafPrefix`



*nodes not to scale

COUNTING LOWER NODES

- Note
  - Leaf keys are already sorted in d_leaf_keys
  - Recall leaf keys are *tile ID*, *lower offset*, ~~*leaf offset*~~
  - Shift them right by 12 bits → lower node keys

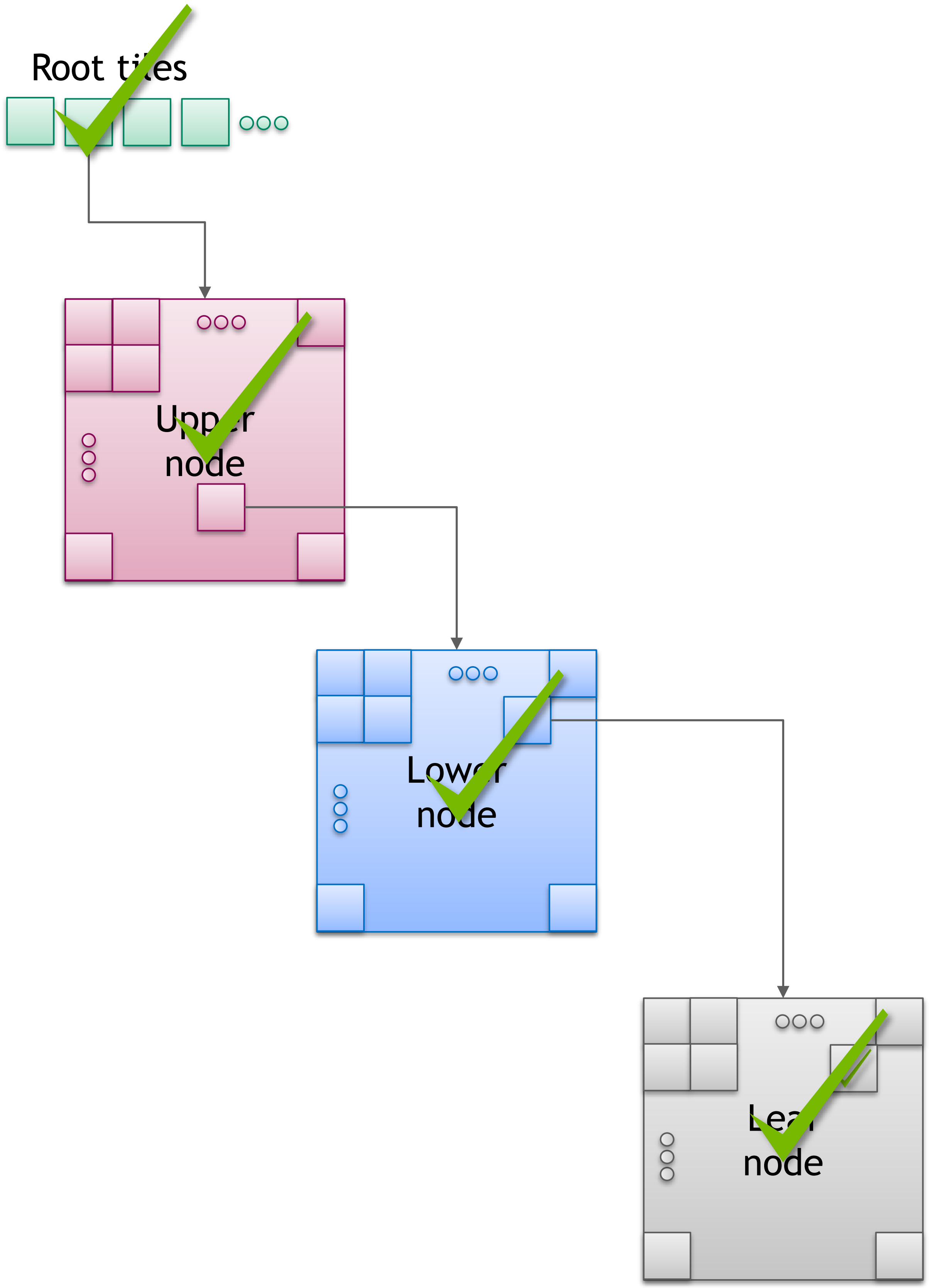


- We don’t need binning at this point, just the number and values of each lower node!
- Unique on d_leaf_keys with a *Right Shift 12 Bits Iterator*, outputs:

d_lower_keys   lower node count

NVIDIA

Done counting the nodes!

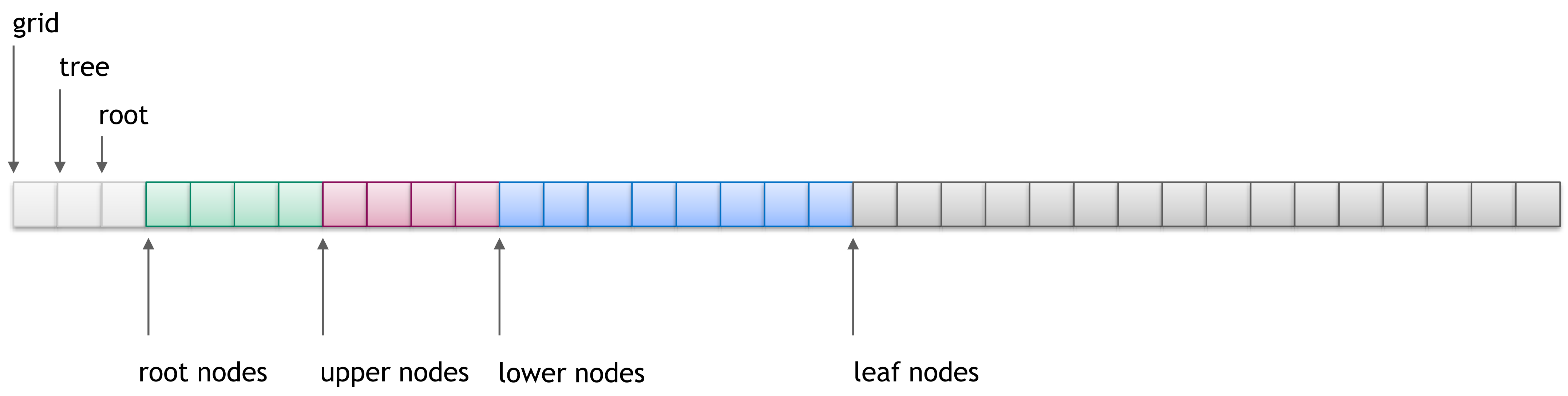


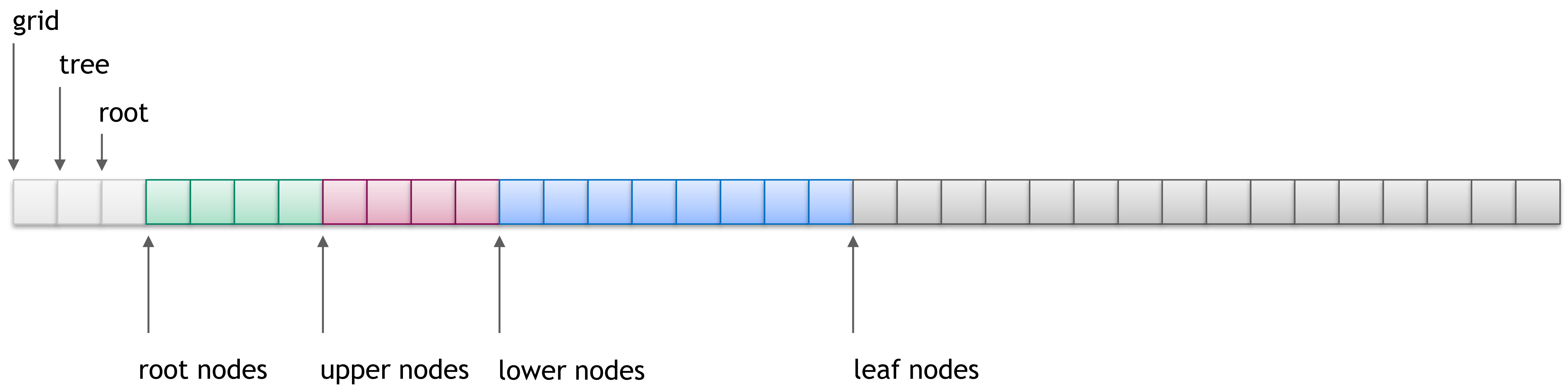
*nodes not to scale

ALLOCATING BUFFER

READY TO ALLOCATE MEMORY

- Now that we have the number of nodes, we can allocate the buffer for the grid
- We are including the seed points as well in the blind data
- At this point we know the place in memory of all the nodes *by ordinal indexing*, eg. the *n*th lower node, but not by spatial coordinates



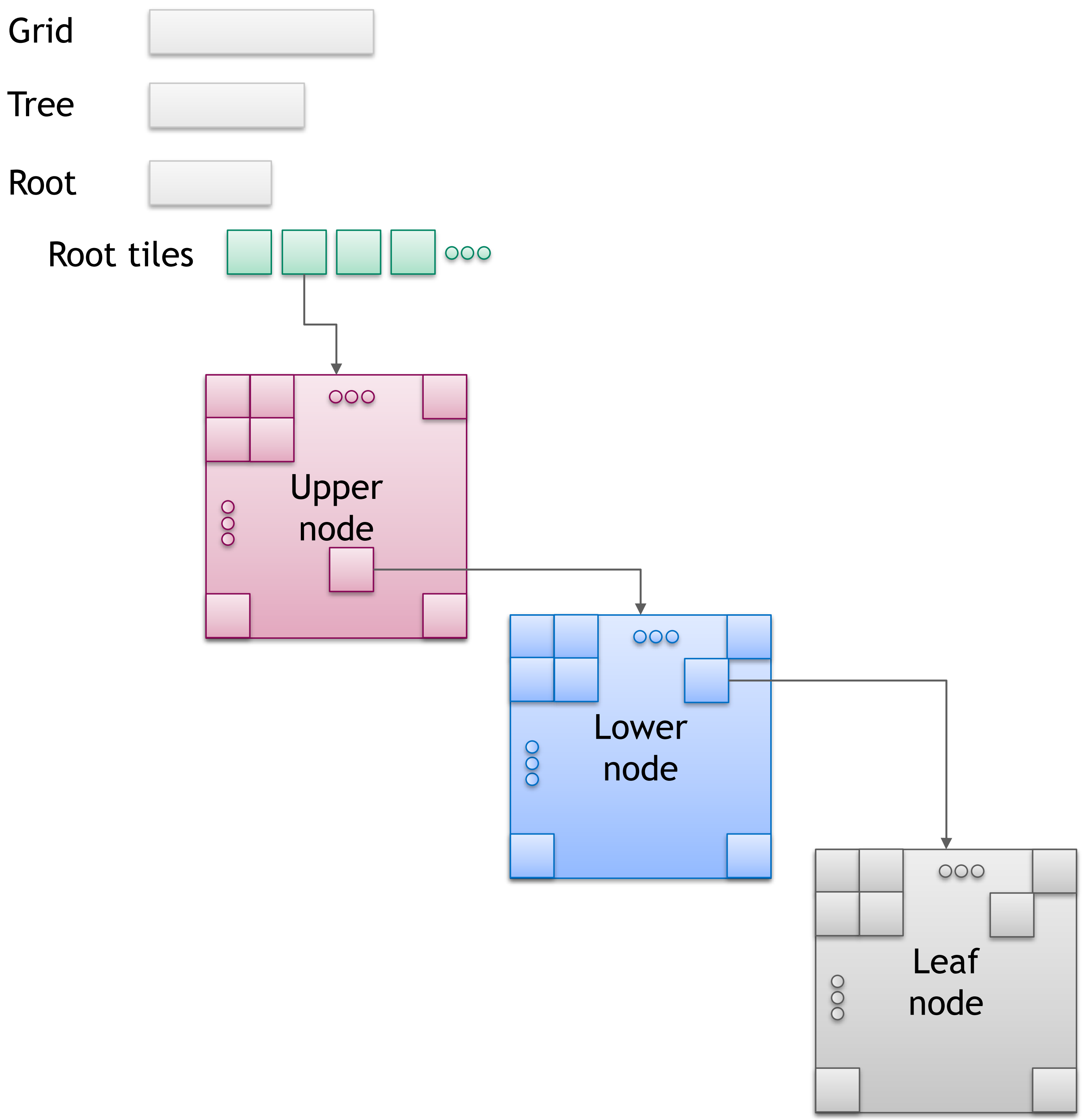


- `NanoUpper<BuildT>& getUpper(int i) const {return *(PtrAdd<NanoUpper<BuildT>>(d_bufferPtr, upper)+i);}`
- `NanoLower<BuildT>& getLower(int i) const {return *(PtrAdd<NanoLower<BuildT>>(d_bufferPtr, lower)+i);}`
- `NanoLeaf<BuildT>&  getLeaf(int i)  const {return *(PtrAdd<NanoLeaf<BuildT>>(d_bufferPtr, leaf)+i);}`
  - E.g. access to `getLower(i)` is valid, if `0<= i < lower_node_count`!
- But we don't know their *spatial* positions!
  - E.g. given *ijk* coordinates, we don't know how to get to that leaf, even though it is allocated.

BUILDING THE TREE

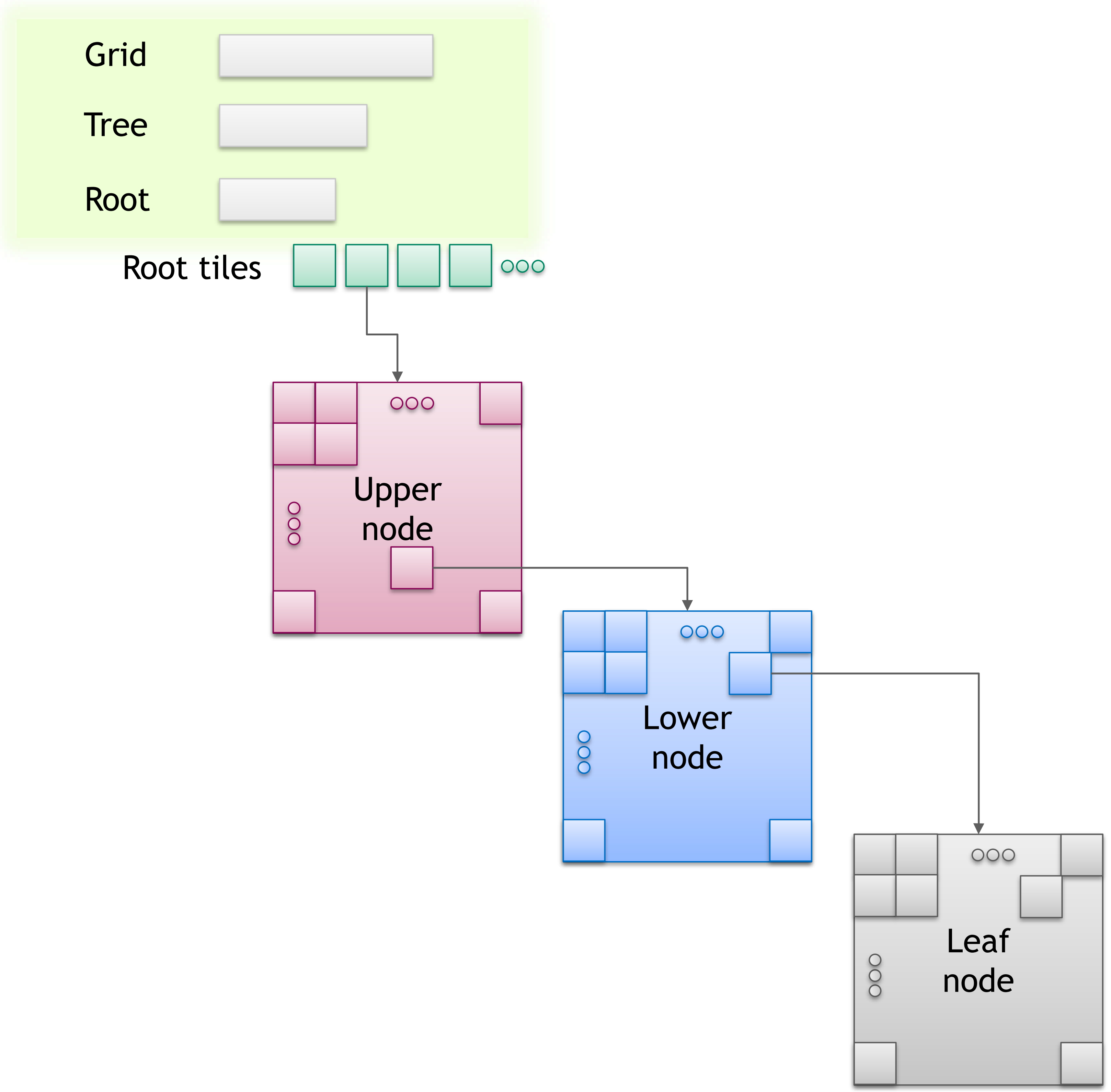
BUILDING THE TREE

- Top-down sweep:
  - Grid, Tree, and Root
  - Upper nodes
  - Lower nodes
  - Leaf nodes
  - Points
- Bottom-up sweep:
  - Computing the Bounding Boxes



GRID, TREE, AND ROOT

- `CudaPointsToGrid::processGridTreeRoot`
- Single-thread kernel
- Straightforward housekeeping



UPPER NODES

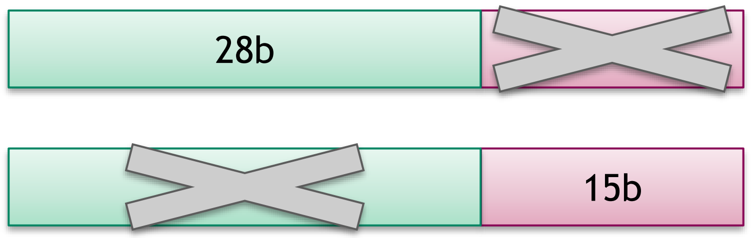
- `CudaPointsToGrid::processUpperNodes`

- Running on # of upper nodes threads:
  - *tid* is `upper node id`. Get the nodes with `getUpper`
  - *Ijk* cords of the upper node: NanoRoot<uint32_t>::KeyToCoord(d_tile_keys [tid]);
  - Records the upper node to the root tile
- Running on (# of upper nodes * 2^15) threads:
  - Zeroing the tables of every upper nodes

LOWER NODES

- `CudaPointsToGrid::processLowerNodes`

- Very similar as before, but
    - const uint64_t lowerKey = d_lower_keys [tid];
    - auto &upper = d_data->getUpper(lowerKey >> 15);
    - const uint32_t upperOffset = lowerKey & 32767u;

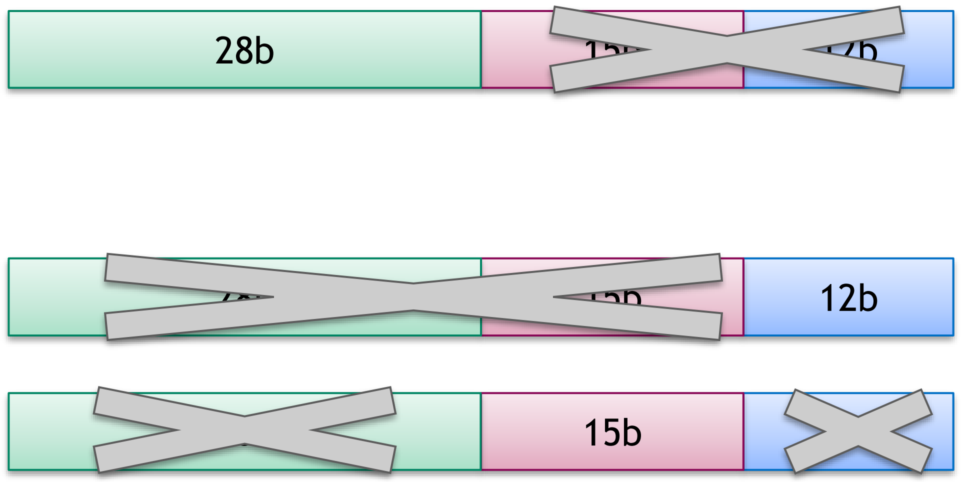


- Needs to use atomic operations to set child mask in parent!
- New kernel launch for resetting the table

LEAF NODES

- `CudaPointsToGrid::processLeafNodes`

- For each leaf node
  - leafKey = d_leaf_keys [tid];
  - tile_id = leafKey >> 27;
  - auto &upper = d_data->getUpper(tile_id);
  - const uint32_t lowerOffset = leafKey & 4095u
  - upperOffset = (leafKey >> 12) & 32767u;



  - Record offset and point count in leaf, if building a point grid
- New kernel launch each active voxel: Record either point ID, or just 1 as a placeholder value

POINTS

Only for point grids!

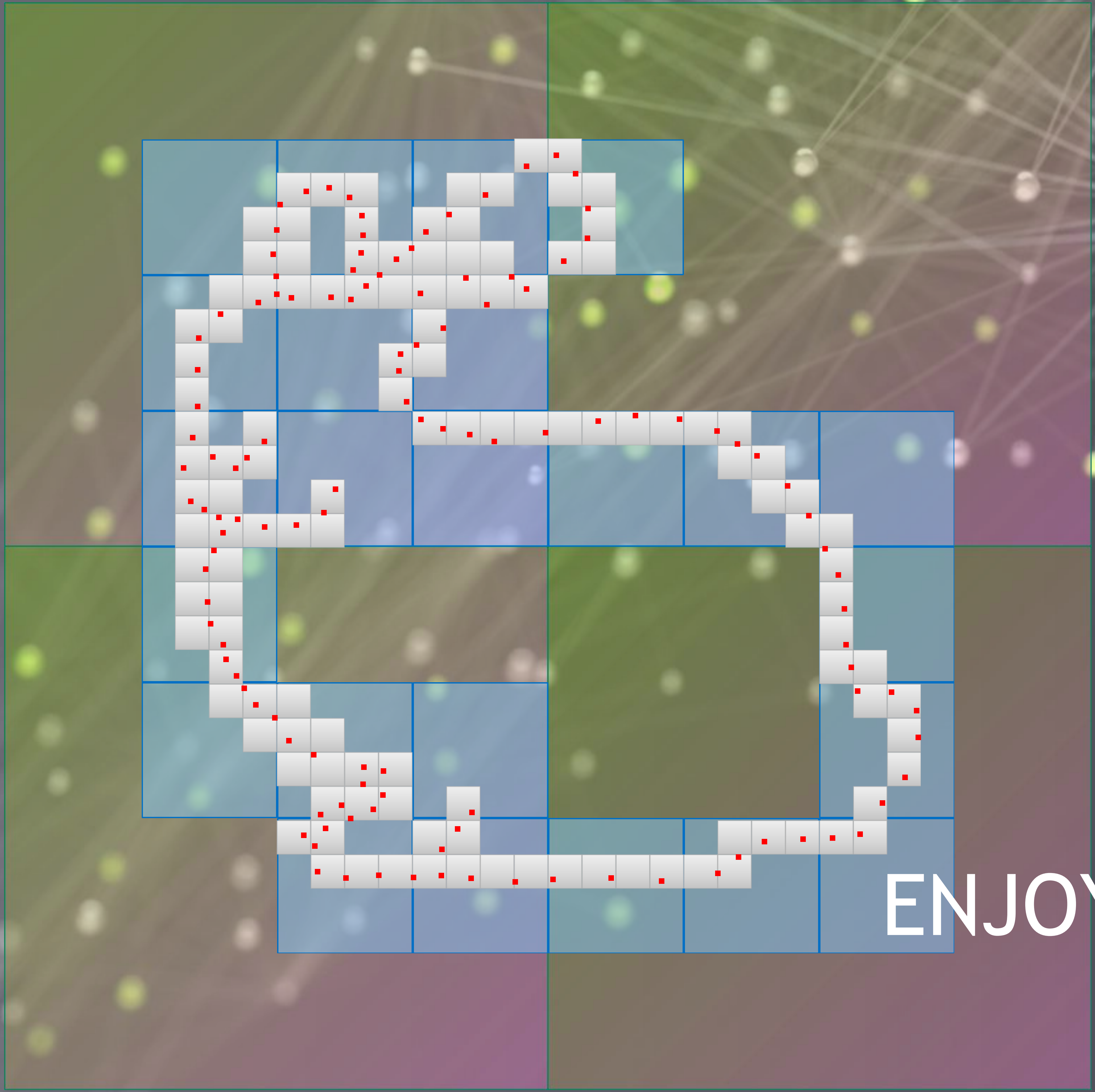
- CudaPointsToGrid<Points>::processPoints

- Copy point IDs or values, based on grid type, into blind data
- Uses d_indx for voxel to point ID lookup

COMPUTING THE BOUNDING BOXES

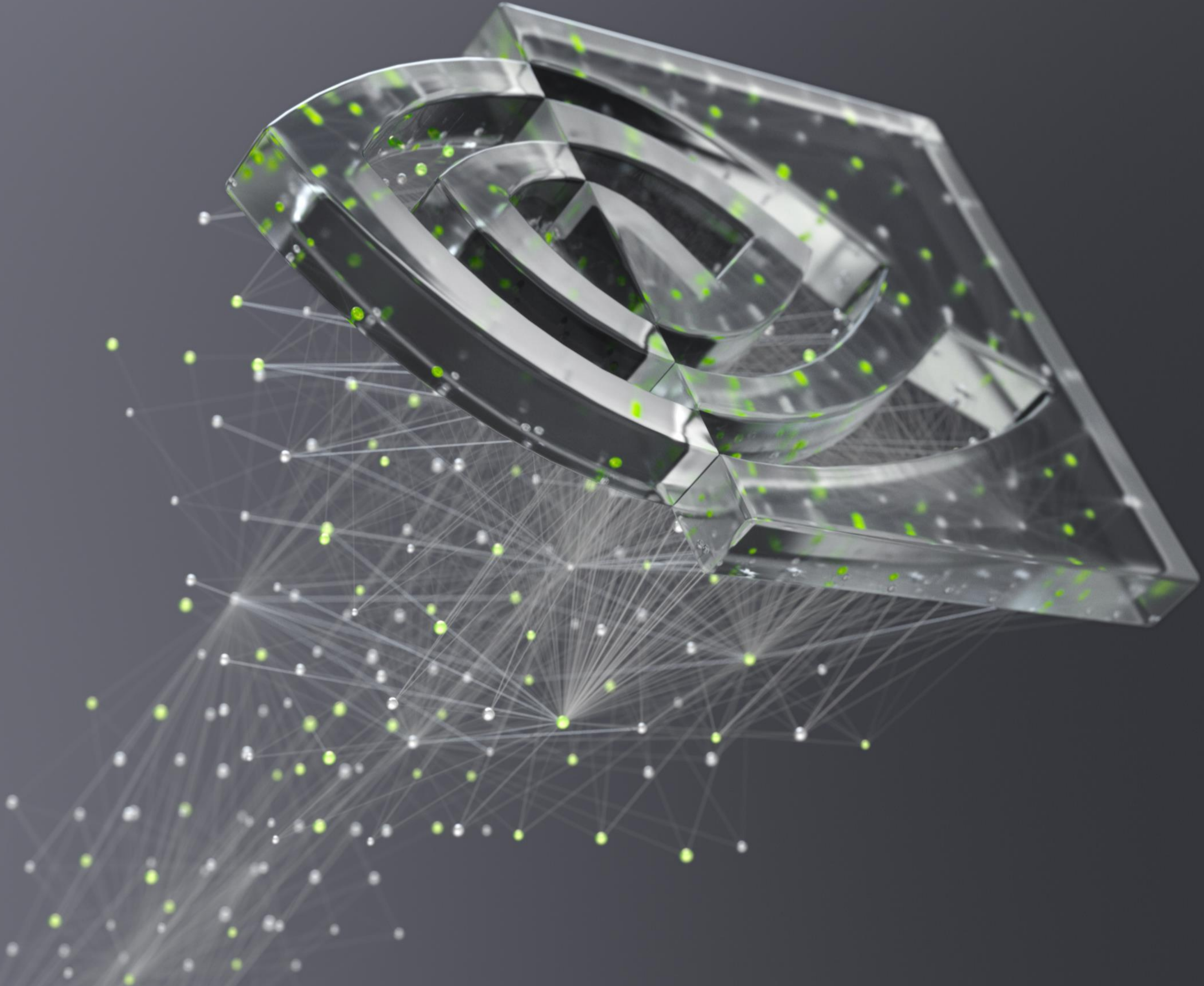
- Lower to upper nodes
- Upper to root nodes
- World space bounding box on grid

- All on the GPU
- Uses `expandAtomic`



ENJOY YOUR BRAND NEW NANOVDB!

THANK YOU!



NVIDIA