



**CSCS**

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre

**ETH** zürich



# Introduction to GPUs in HPC

**CSCS Summer School 2025**

**Andreas Jocksch, Prashanth Kanduri, Radim Janalik & Sebastian Keller**

# Implementing non-trivially parallel problems on GPUs: tree data structures

# Why focus on tree data structures?

Tree-based data structures have many uses in HPC. That includes binary radix trees, quadtrees and octrees:

- Neighbor searches for particles in 2D/3D point clouds
- Collision detection in 3D computer graphics
- Adaptive mesh refinement (AMR)
- Domain decomposition to run on multiple compute nodes
- Fast Multipole Method (FMM) for N-body problems

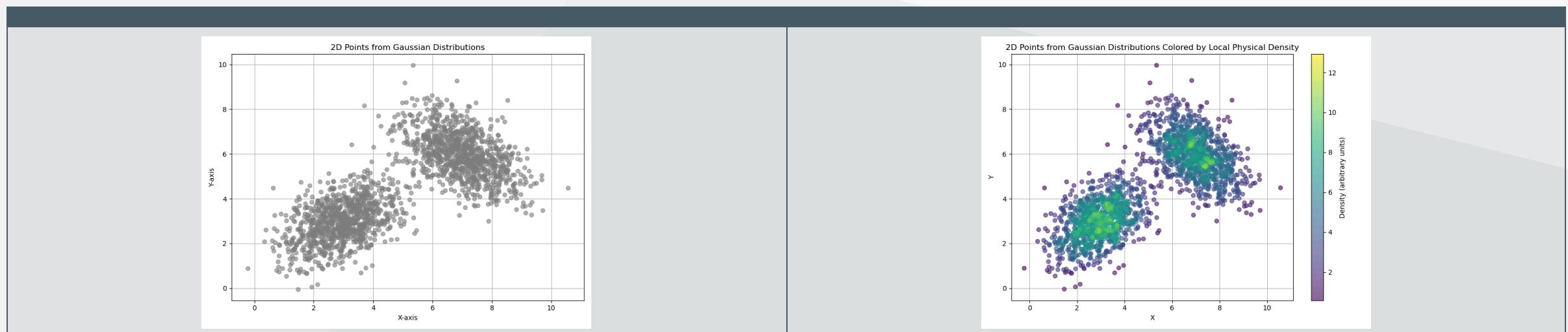
**Many naive parallel approaches to trees that work on few-core CPUs do not translate well to GPUs.**

# First an example: Neighbor search

- Finding data which are "close" to each other is a common problem in science.
- Machine Learning: Finding groups of similar characteristics for classification.

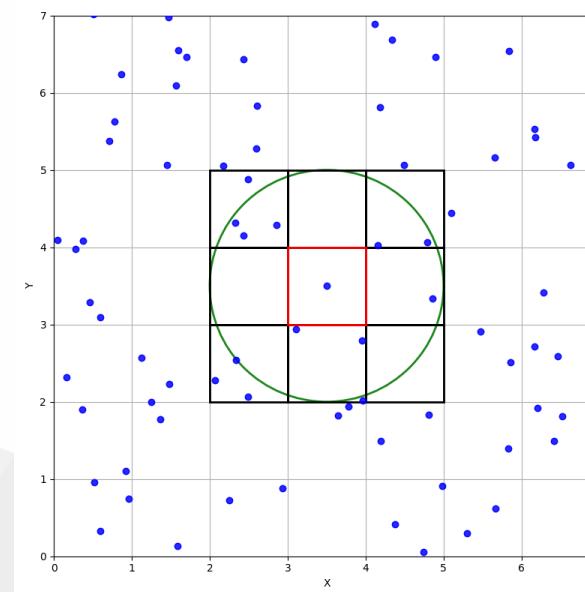


- Physics: Density estimates in particle-based models based on the distribution of  $N$  closest neighbors.



# How could we find the neighbors of all the particles within a cutoff?

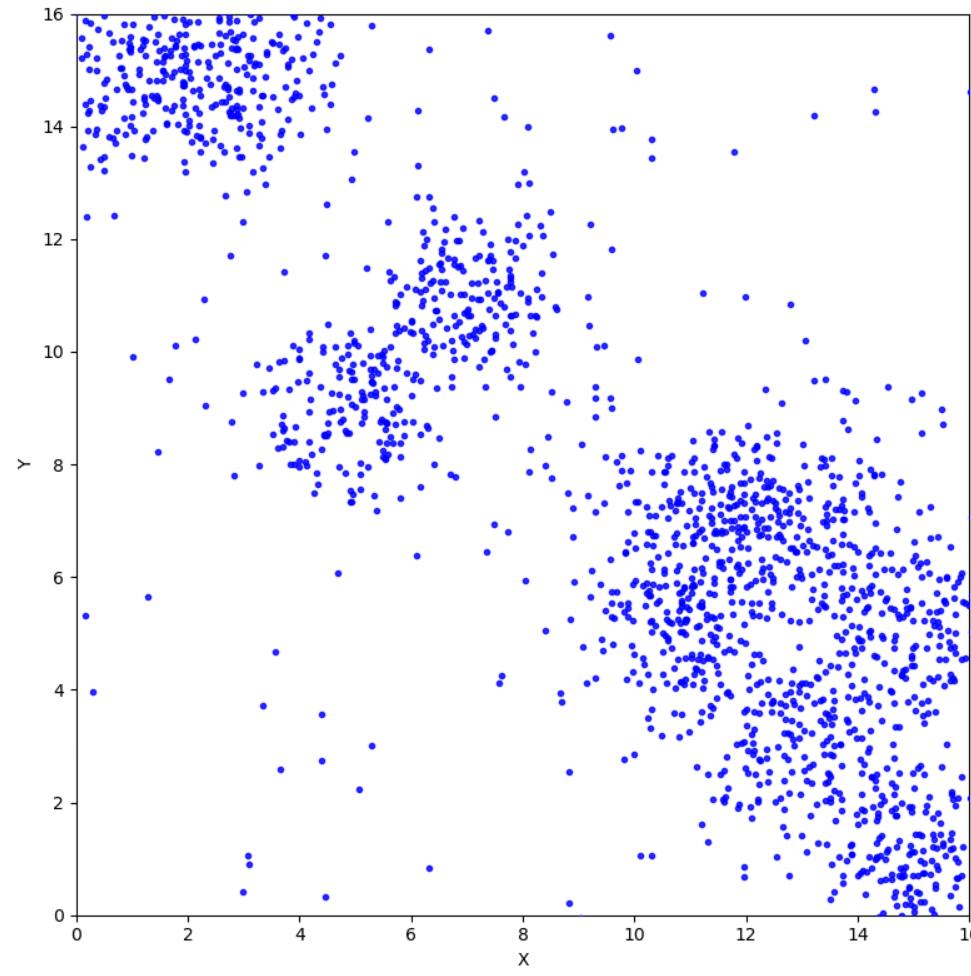
1. For each particle: compute the distance to each other particle, accept/reject.  $O(n^2)$
2. Could use Verlet lists (popular in molecular dynamics) to reduce  $n \rightarrow m$  with a fixed resolution grid, but we need to partition the particles first.  $O(mn)$



3. Hierarchical space partitioning (kd-tree, ORB, octree, etc.).  $O(n \log n)$

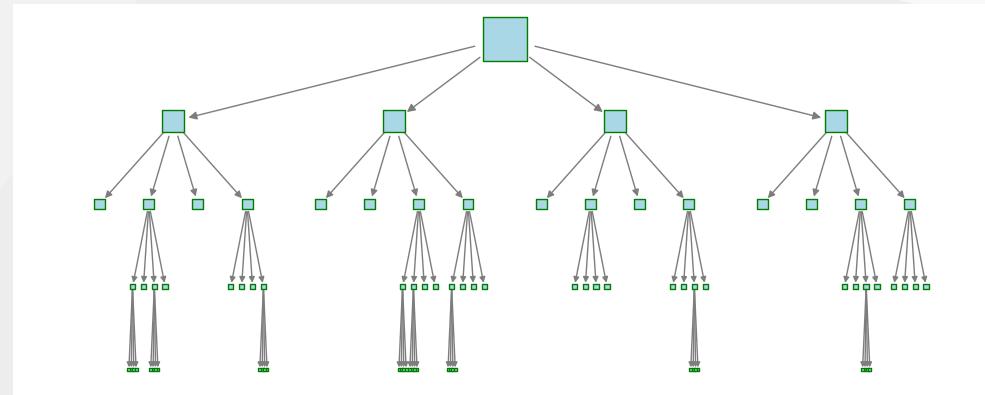
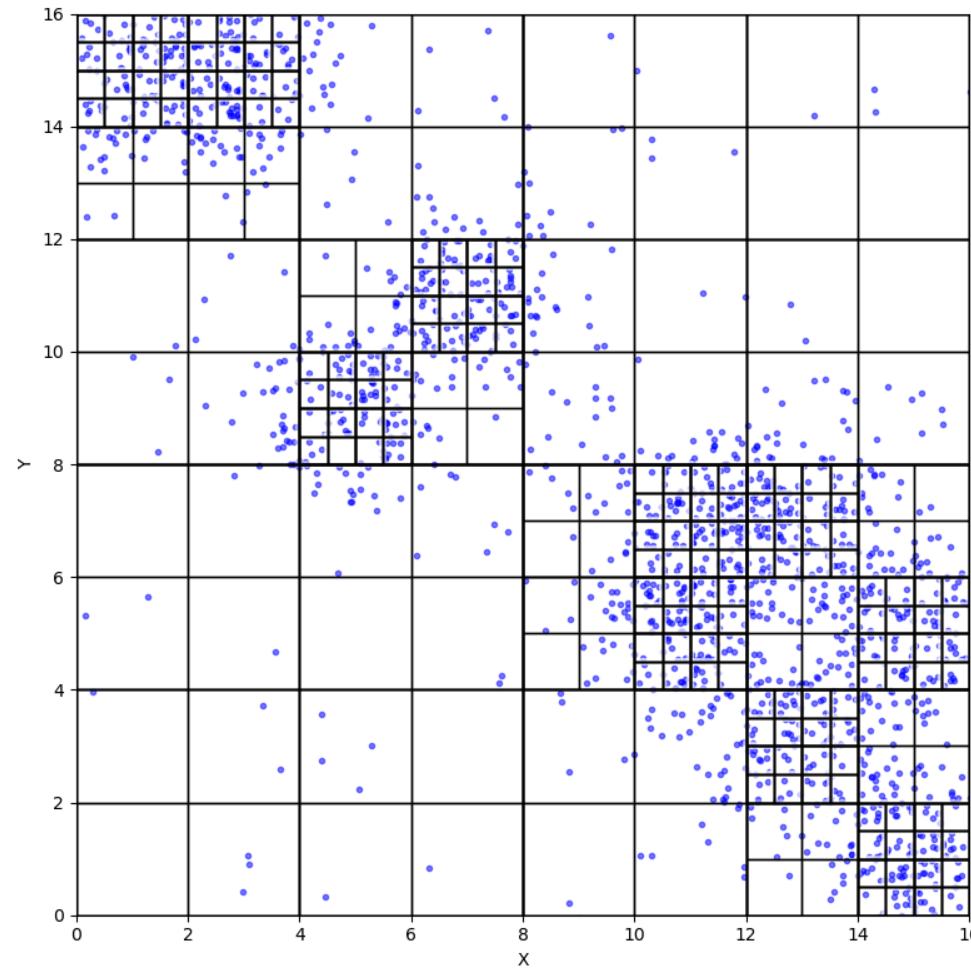
# Hierarchical space partitioning

In 2D, let's consider the Quadtree:



# Hierarchical space partitioning

In 2D, let's consider the Quadtree. The Octree is the natural 3D extension.



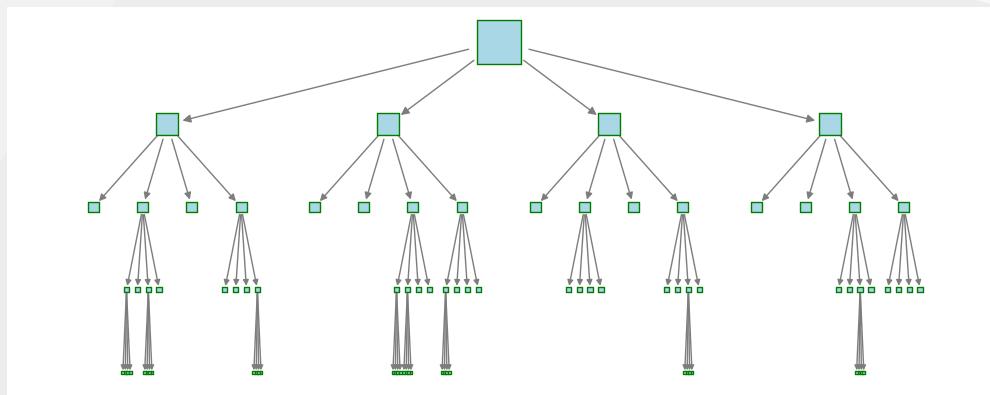
# Tree Data Structure

A straight-forward tree data layout with pointers:

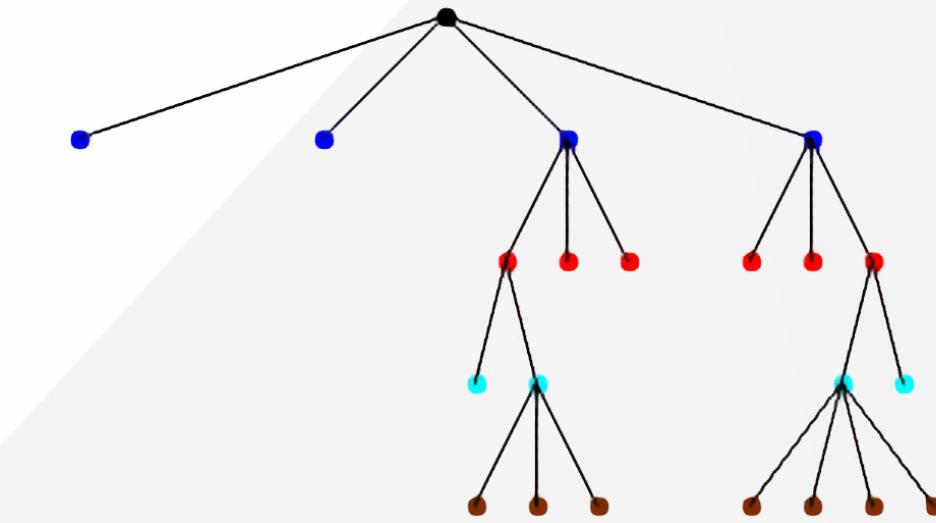
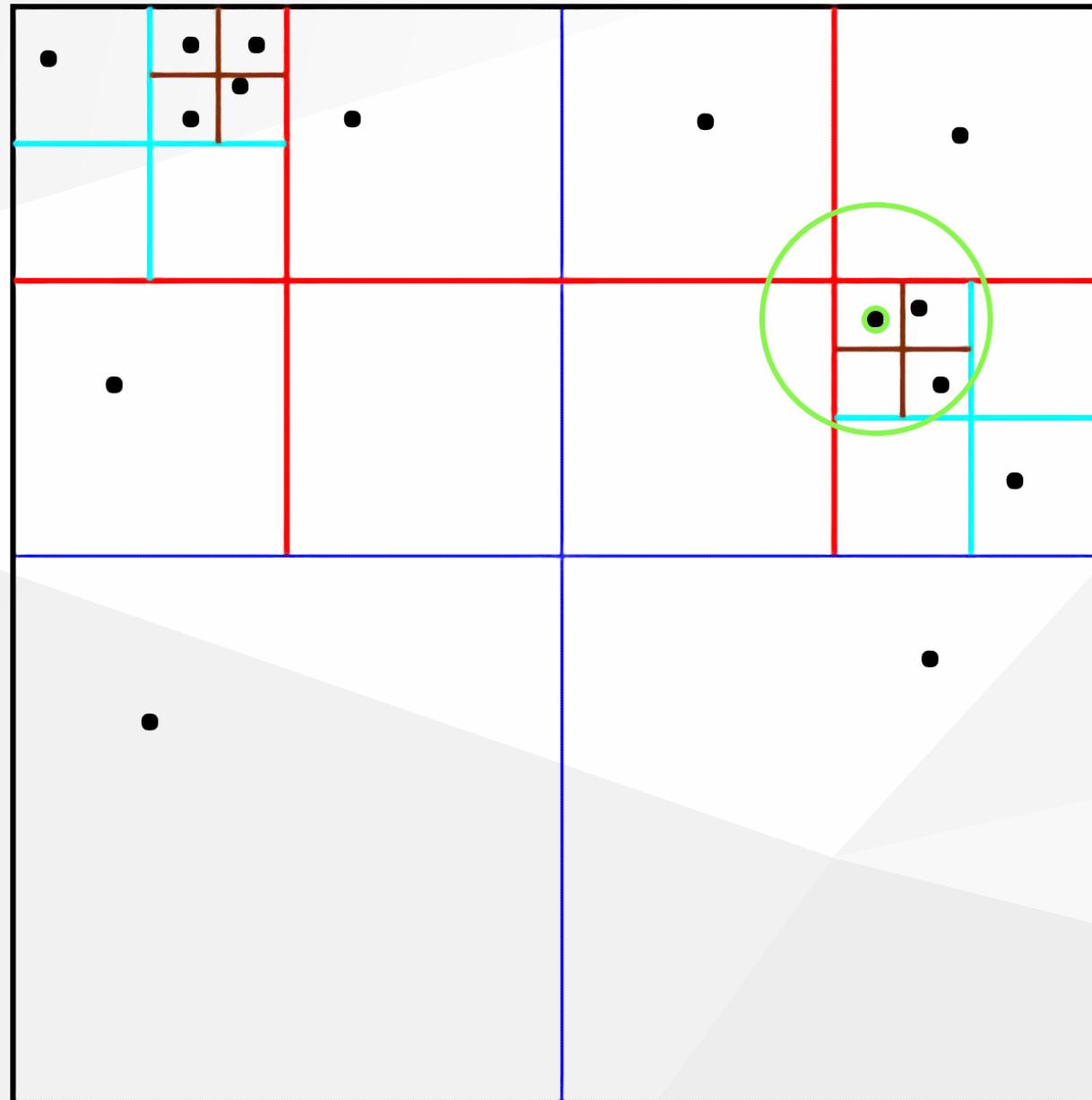
```
struct TreeNode
{
    // connectivity
    int numChildren;
    TreeNode* children[numChildren];
    TreeNode* parent;

    // location in space
    std::array<double, 3> nodeCenter;
    std::array<double, 3> nodeSize;

    // additional node properties, e.g.
    int numParticles; // etc...
};
```



# Neighbour Search Intuition



# All Neighbours within Radius $R$ using Trees

Basic algorithmic idea:

For each particle at position  $r$ ,

1. Push the root node onto a work stack.
2. Pop the top most node from the work stack.
3. Is the node completely within a sphere of radius  $R$  at  $r$ ?
  - If yes, accept all particles. Goto 2.
4. Is the node a leaf node?
  - If yes, accept each particle in the node for which  $|R - r| < R$ . Goto 2.
5. Does the node's bounding box intersect with a sphere of radius  $R$  at  $r$ ?
  - If yes, push all the child nodes onto the stack.
6. Goto 2.

Can this be efficiently implemented on GPUs? Answer: depends on data layout.

# What doesn't work

Let's assume the following parallelization strategy:

one GPU thread -> neighbor search for one particle.

A straight-forward tree data layout with pointers won't work on the GPU:

- **Tree node** pointers do not ensure that sequentially accessed nodes are local to each other in memory.
- **Particles** in a leaf node are also not guaranteed to be grouped in memory.
- Both scenarios cause memory to be accessed very inefficiently by jumping around.
- If particles of two consecutive GPU threads are far away from each other, paths down the tree will diverge early, causing thread(warp) divergence
- Putting all properties together in a node can lead to poor cacheline usage.

# What does work

For the tree:

- Storing tree nodes in arrays (linear buffer)
- A separate array for each node property (-> **struct of arrays** instead of array of structs)

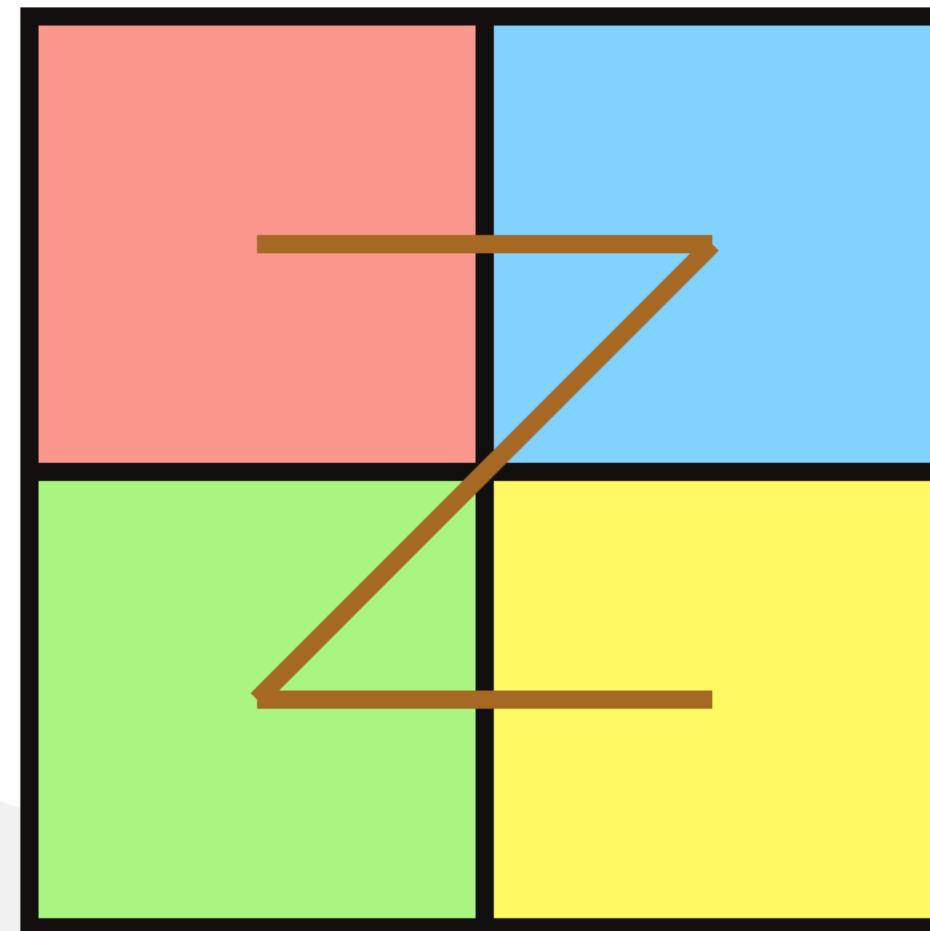
For the particles:

- Particles close in space should be close in memory. This can be achieved for example with **Space Filling Curves (SFCs)**, e.g. Z-curve or Hilbert curve to encode the location in space

References: [Binary radix trees](#)

# Space Filling Curves: Intuition

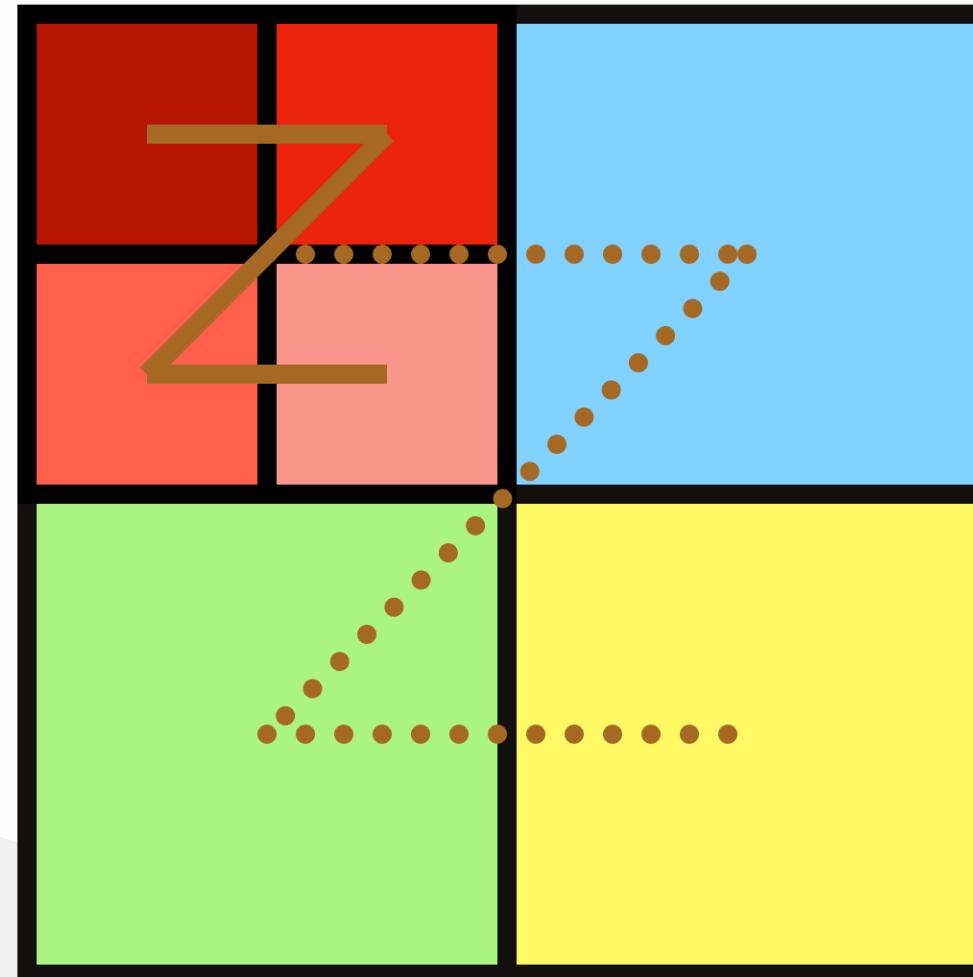
SFCs are mathematical functions that map points in 2D/3D ...



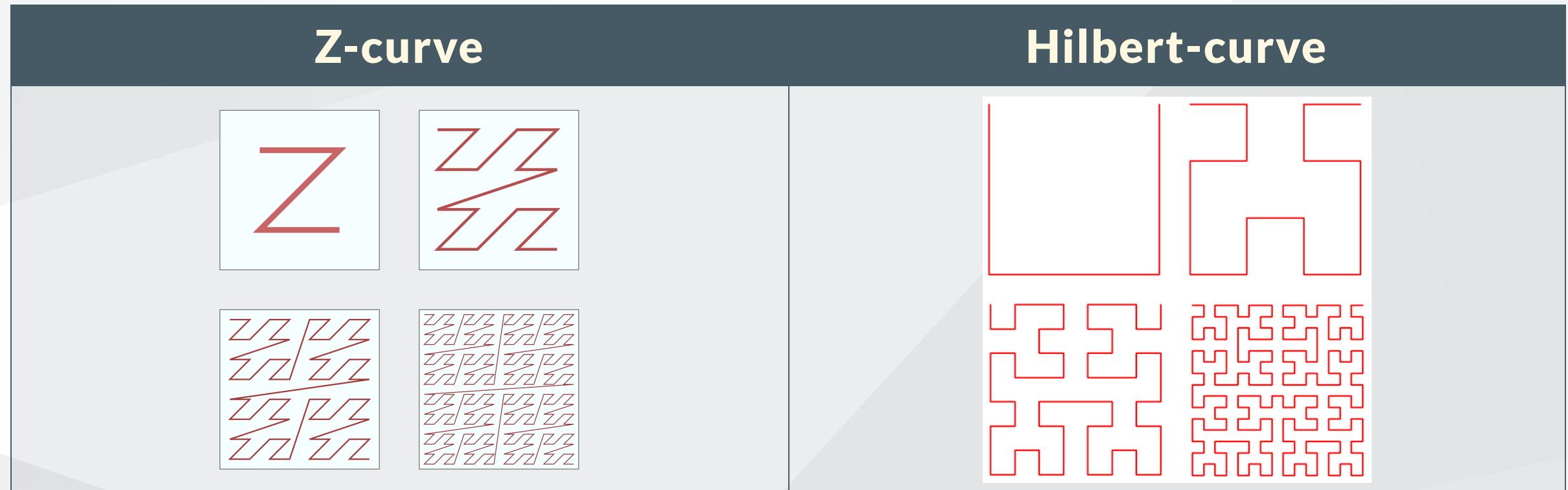
... into the unit interval  $[0, 1]$ . Areas/volumes into sub-intervals.



# Space Filling Curves: self-similarity



# Space Filling Curves: Discretization



On a computer, **areas/volumes** are represented as **grids** with  $M^2$  or  $M^3$  grid-points and the **1D interval or key-space** is  $[0, \dots, M^{2,3} - 1]$ .

- SFC enumerates the grid-points
- $M$  has to be a power of 2:  $M = 2^L \rightarrow$  key-space fits in  $2L$  or  $3L$  bits
- Grid points align with quad/octree cells at tree-depth  $L$
- Higher tree nodes correspond to ranges in the key-space

# Data layout: particles

array	type	dimensions	description
x	FP32/64	[ numParticles ]	x-coordinates
y	FP32/64	[ numParticles ]	y-coordinates
z	FP32/64	[ numParticles ]	z-coordinates
keys	uint64	[ numParticles ]	SFC keys

- particle `x, y, z` arrays are ordered such that `keys` is sorted

# Data layout: octrees

Node geometry:

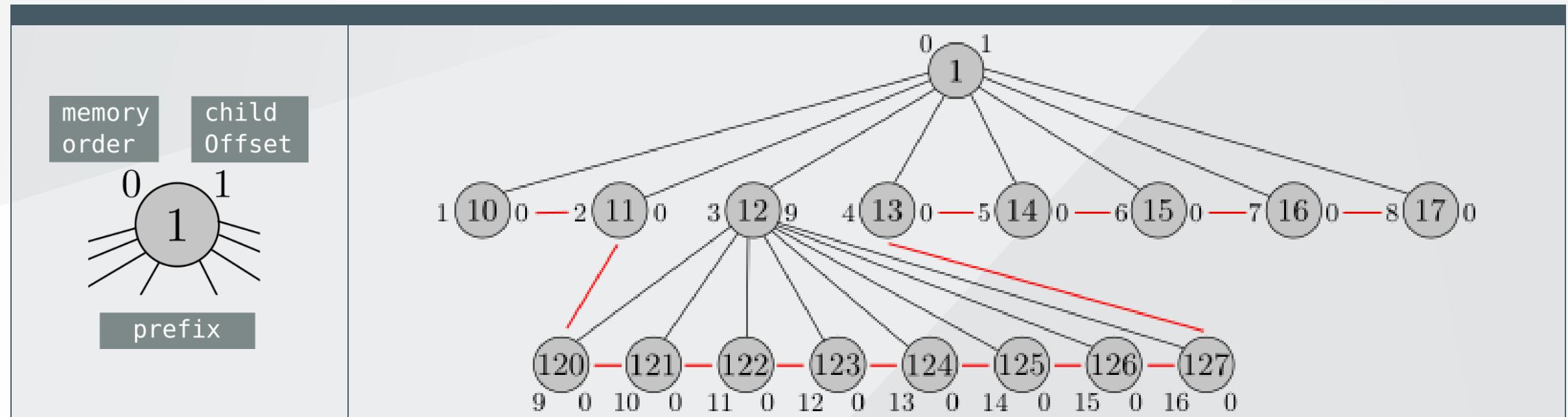
array	type	dimensions	description
prefixes	uint64	[numNodes]	node SFC key
nodeCenters	Vec3<FP32/64>	[numNodes]	3D center coordinate
nodeSizes	Vec3<FP32/64>	[numNodes]	3D node size

- `prefixes` can be decoded into `nodeCenters` and `nodeSizes`

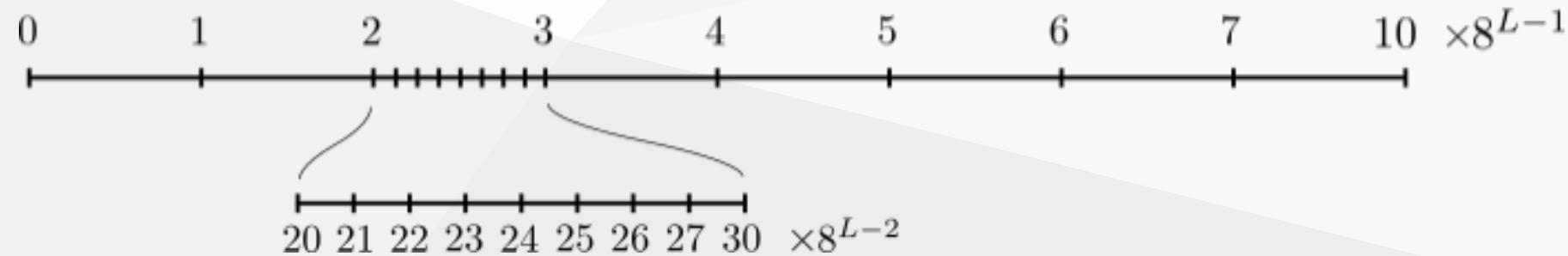
Node connectivity:

array	type	dimensions	description
childOffsets	int32	[numNodes]	first child node
internalToLeaf	int32	[numNodes]	enumeration of leaf nodes
layout	uint32	[numLeafNodes]	index of first particle in x, y, z

# Data layout: octrees



Just the leaf nodes as they appear in `internalToLeaf` (the red line above):



Example: particles of node 9: `internalToLeaf[9] = 2`, so from `layout[2]` to `layout[3]`

# Octree mini-app repository

We provide a mini-app repository with the following features:

- **Octree construction** from 3D particle point clouds on CPUs and GPUs
- Portable **neighbor search** implementation with depth-first traversal
- Highly optimized explicitly warp-convergent neighbor search with breadth-first traversal

We use the mini-app as a basis for the next exercise, but we won't have time to work through all features. **You're invited to explore!**

For further study, you may also read the paper below.

References: [Cornerstone octree](#)

# Important porting considerations

- Memory access pattern: consecutive threads need to load consecutive addresses in memory
- SIMT and warp-divergence: threads in a warp need to execute the same instructions, branching divergence leads to loss of performance
- CUDA offers several technical features to aid in parallelization and optimization:
  - shared memory and block synchronization
  - intra-warp communication
  - cooperative groups
  - streams and graphs
- Parallel building blocks often serve as a stepping stone. These can be extremely efficient in parallel:
  - reductions ( $1 + 2 + 3 + 4 \rightarrow 10$ )
  - prefix sums (Blelloch scan) ( $1 + 2 + 3 + 4 \rightarrow \{1,3,6,10\}$ )
  - sorting (radix sort)

# Take-home Message

## **Massive parallelism may need new algorithms**

- Often, a fundamentally new algorithm is needed to expose parallelism
- If your app is inherently sequential or the data layout is not suitable, the technical features of CUDA won't allow you to run efficiently!

# Exercise 1: tree traversal for neighbor searching

1. Implementation of neighbor search on CPUs ( `findneighbors.hpp` ).
  - compile with `nvcc -std=c++17 -O3 -Xcompiler -fopenmp neighbor_search.cu`
  - follow the hints given in the file
  - can you make the `all2all` test pass?
2. Implementation of neighbor search on GPUs
  - implement the CUDA kernel in `neighbor_search.cu`
  - which CPU functions can we also use on the GPU?
  - can you make the CPU reference test pass?
  - how much speed-up do you get for 2 million particles?
  - activate the warp-level optimized version, how much faster is it?

# Exercise 2: sorting particles along the Space-Filling-Curve

You are provided with two programs: `octree.cpp` and `octree.cu`. They both construct an octree for the same distribution of 3D particles.

- `octree.cpp` is implemented on CPUs and is fully working.
- In `octree.cu`, the function that sorted the particles and their SFC keys is missing, this has to be implemented.
- You can achieve this by porting the provided CPU version step-by-step

