

---

# Massively Parallel Computing with Cuda

- Searching -

**Hendrik Lensch**  
**Robert Strzodka**

# Today

---

- **Last lecture**
  - Data Structures
- **Today**
  - Parallelization Speedup – Amdahl's law
  - Parallel Searching
  - Parallel Acceleration Structures

# Measures of Efficiency

---

- **Sequential task A of size n.**
- **$t_S(n)$  is the number of sequential steps required to execute A for n elements**

# Measures of Efficiency

---

- **Given the parallel task A**
- **$t_A(n)$  is the parallel time (depth) of A**
  - the maximum number of timesteps for any of the parallel processors
- **$w_A(n)$  is the parallel work of A**
  - overall number of operations performed summed over all processors
- **$p_i$  is the number of processors needed to execute step i in constant time**

$$w_A(n) = \sum_{i=0}^{t_A(n)} p_i(n)$$

- **$p_A$  = maximum number of processors required**
- **cost**  $c_A(n) = p_A(n)t_A(n)$

# Efficiency

---

- A weak requirement for an efficient parallel algorithm A

$$\frac{t_A(n)}{t_S(n)} \rightarrow 0 \quad \text{for } n \rightarrow \infty$$

# Work and Time Optimality

---

- A parallel algorithm A is work-optimal if

$$w_A(n) = O(t_S(n))$$

- A parallel algorithm A is time-optimal if any other parallel algorithm would require at least  $\Omega(t_A(n))$  time steps
- A parallel algorithm A is work and time-optimal if any other work-optimal algorithm would require  $\Omega(t_A(n))$  time steps

# Work Efficiency

---

- Let **S** be either an optimal or a currently best known sequential algorithm for problem **P**
- A parallel algorithm **A** is work-efficient for **P** if

$$w_A(n) = t_s(n) \cdot O(\log^k(t_s(n)))$$

- for  $k \geq 1$

# Brent's Theorem

---

- An PRAM algorithm A which runs in  $t_A(n)$  time steps and performs  $w_A(n)$  work can be implemented to run on a p-processor PRAM in

$$t_A(p, n) = O\left(t_A(n) + \frac{w_A(n)}{p}\right)$$

# Speedup

---

- **relative to best known sequential algorithm S**

$$\frac{t_S(n)}{t_A(n)}$$

- **absolute speedup for a given number of processors**

$$SU_{abs} \frac{t_S(n)}{t_A(p, n)} \leq p \frac{t_S(n)}{c_A(n)} \leq p$$

- **relative speedup**

$$\frac{t_A(1, n)}{t_A(p, n)}$$

# Amdahl's Law

---

- Split the problem  $A$  into the part  $P$  that can be parallelized and the fraction  $(1-P)$  where no gain in parallelization is achieved
- The maximum speedup is

$$\frac{t_S(n)}{t_{(1-P)}(n) + \frac{t_P(n)}{p}}$$

---

# Parallel Searching

# Parallel Searching Topics

---

- **Searching for a single element in parallel**
- **Parallel search for multiple elements (batch)**
- **Parallel building of index structures**

# Naïve Search for a Single Element

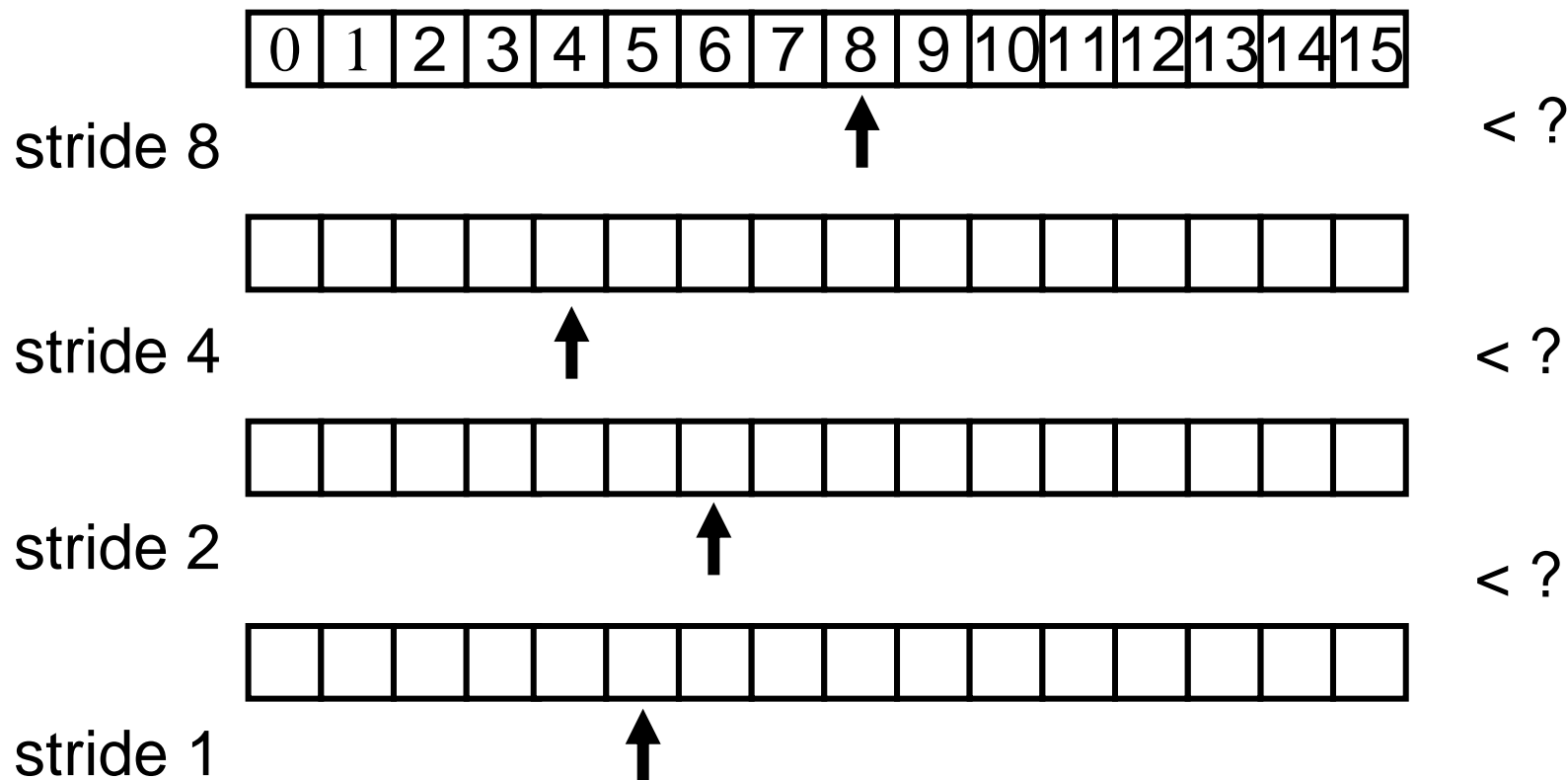
---

- **Distribute array to  $p$  processors**
- **When the element is found the thread writes index to global memory**
- **Complexity:  $O(n/p)$**

# Binary Search

---

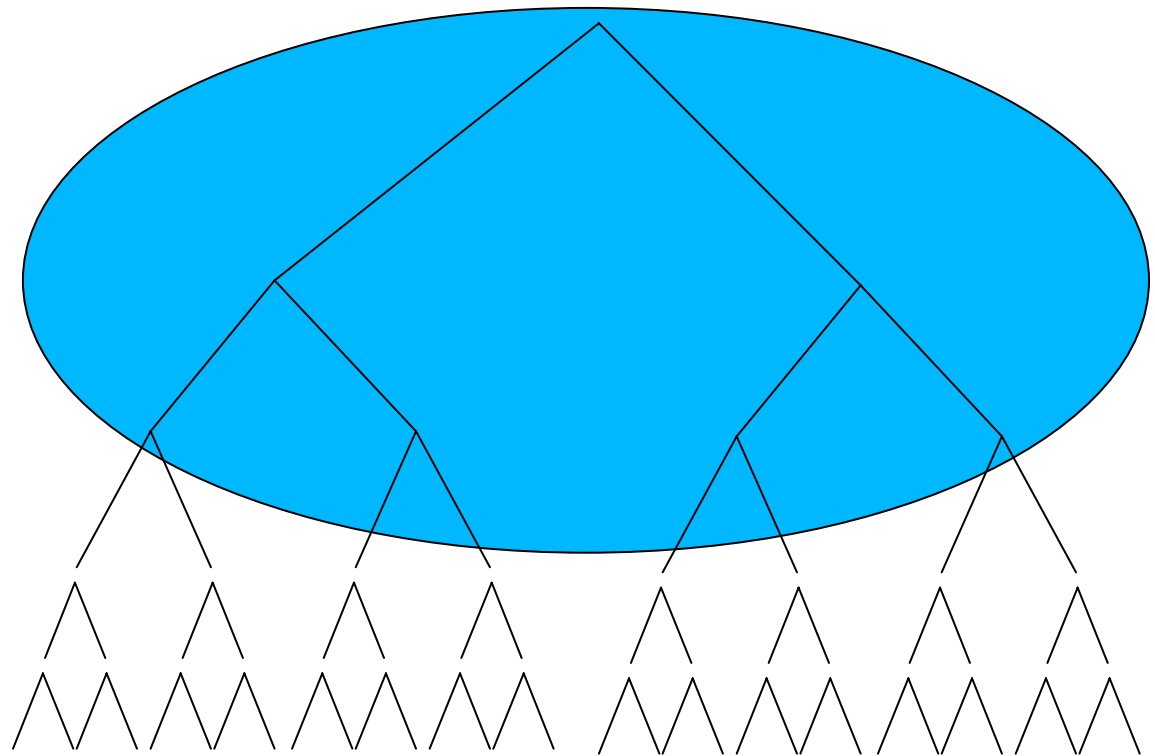
- For ordered lists only
- Determines position where to insert the element, e.g. 5
- $O(\log_2)$



# Batch Binary Search

---

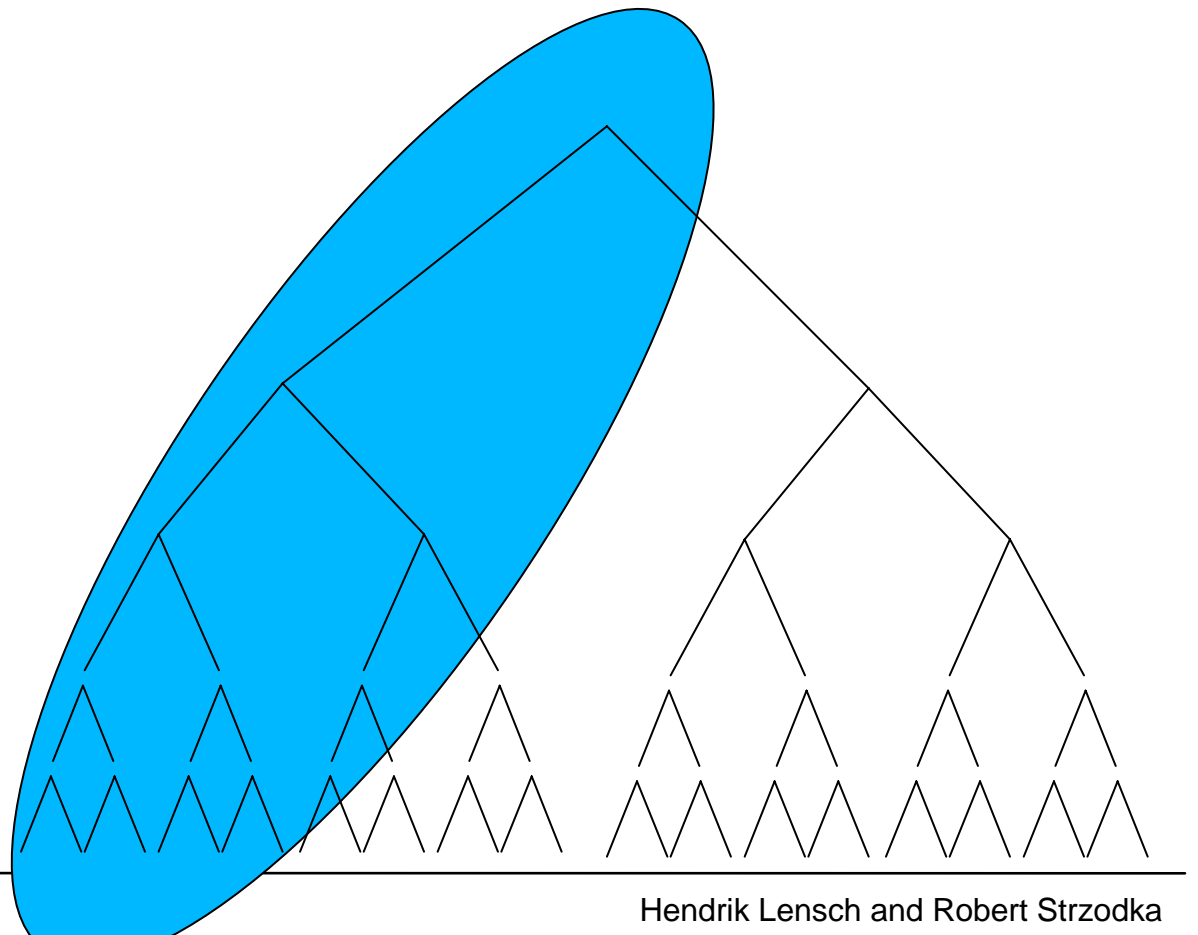
- **N times binary searches performed in parallel**
- **Top of the hierarchy almost always shared**
- **Performance heavily depends on data locality**



# Batch Binary Search

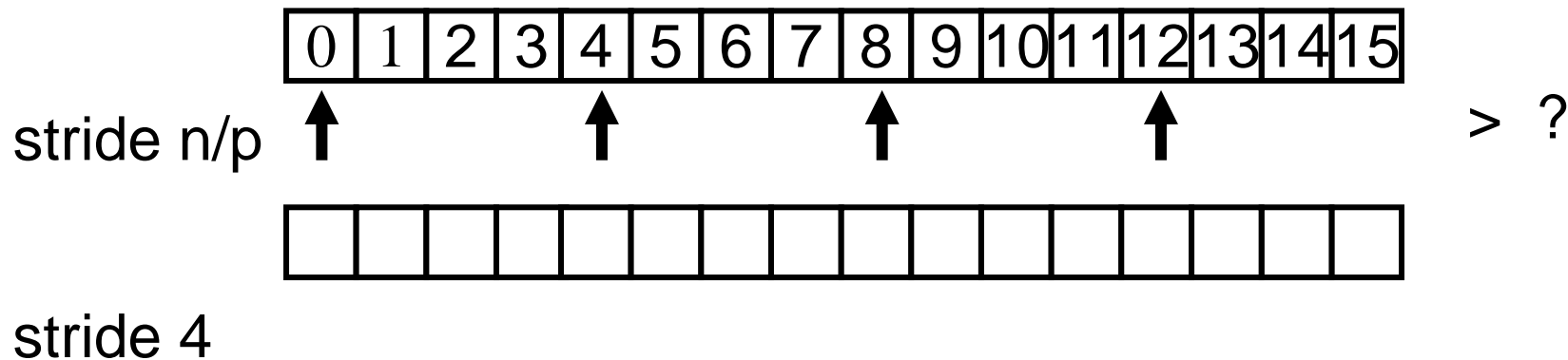
---

- **N times binary searches performed in parallel**
- **Top of the hierarchy almost always shared**
- **Performance heavily depends on data locality**



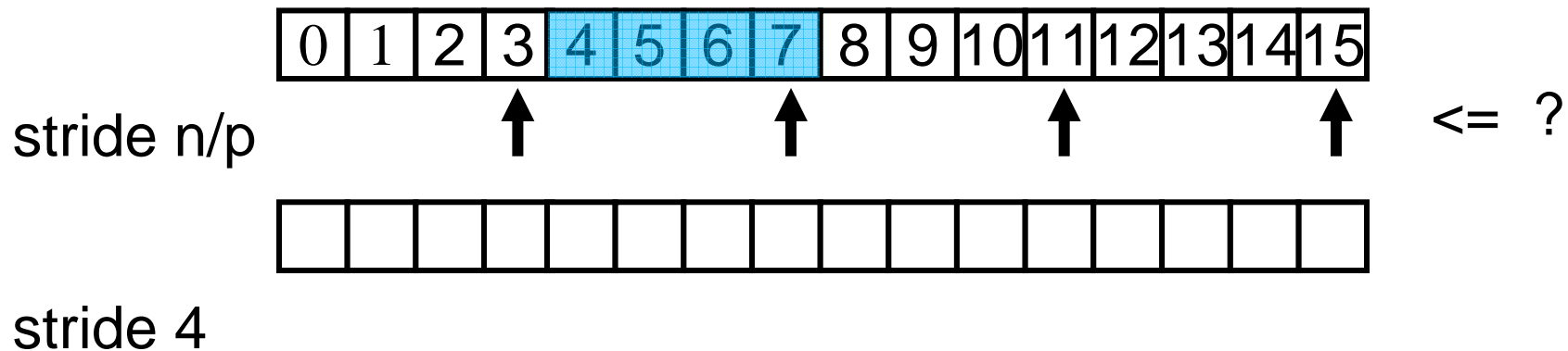
# P-ary Search

- Each processor checks if the element is in its interval
- Stores interval to global memory
- Two comparisons necessary
- $O(\log_p)$



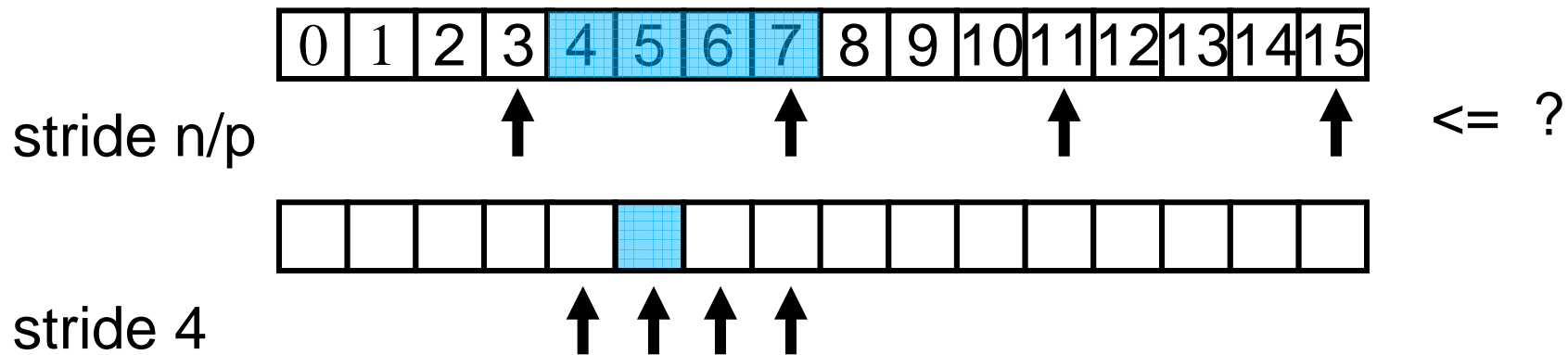
# P-ary Search

- Each processor checks if the element is in its interval
- Stores interval to global memory
- Two comparisons necessary
- $O(\log_p)$



# P-ary Search

- Each processor checks if the element is in its interval
- Stores interval to global memory
- Two comparisons necessary
- $O(\log_p)$  (speedup ?)



---

# Building Acceleration Structures in Parallel

# Sorting

---

- **Given unordered list**
- **Use radix/bitonic/... to output ordered list**

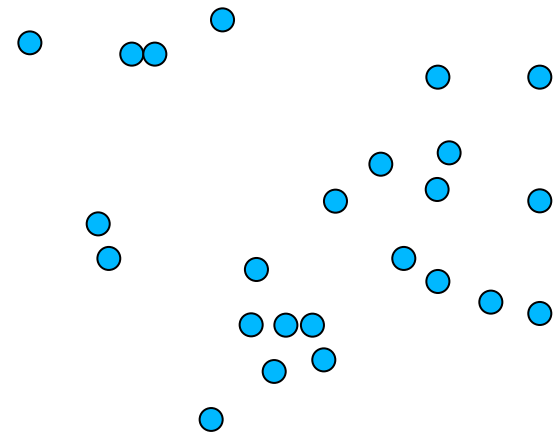
---

# Nearest Neighbor Computations

# K-Nearest Neighbor

---

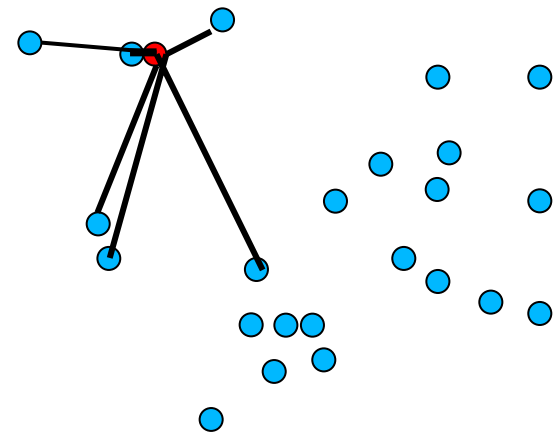
- **Required for density estimations**
- **Task: Report the  $k$  nearest neighbors (kNN) and then compute weight them with a kernel**
- **e.g. photon density estimation:**
  1. Gather the  $N$  nearest photons
  2. Let  $S$  be the sphere that contains these  $N$  photons.
  3. For each photon, divide the amount of flux (real photons) that the photon represents by the area of  $S$  and multiply by the BRDF applied to that photon.
  1. The sum of those results for each photon represents total surface radiance returned by the surface intersection in the direction of the ray that struck it.



# K-Nearest Neighbor

---

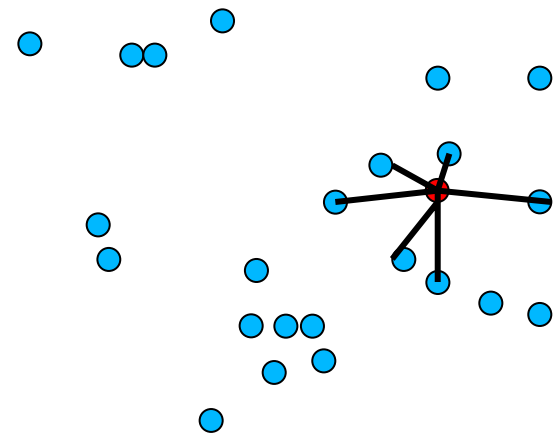
- **Required for density estimations**
- **Task: Report the  $k$  nearest neighbors (kNN) and then compute weight them with a kernel**
- **e.g. photon density estimation:**
  1. Gather the  $N$  nearest photons
  2. Let  $S$  be the sphere that contains these  $N$  photons.
  3. For each photon, divide the amount of flux (real photons) that the photon represents by the area of  $S$  and multiply by the BRDF applied to that photon.
  1. The sum of those results for each photon represents total surface radiance returned by the surface intersection in the direction of the ray that struck it.



# K-Nearest Neighbor

---

- **Required for density estimations**
- **Task: Report the  $k$  nearest neighbors (kNN) and then compute weight them with a kernel**
- **e.g. photon density estimation:**
  1. Gather the  $N$  nearest photons
  2. Let  $S$  be the sphere that contains these  $N$  photons.
  3. For each photon, divide the amount of flux (real photons) that the photon represents by the area of  $S$  and multiply by the BRDF applied to that photon.
  1. The sum of those results for each photon represents total surface radiance returned by the surface intersection in the direction of the ray that struck it.



# Brute-force kNN

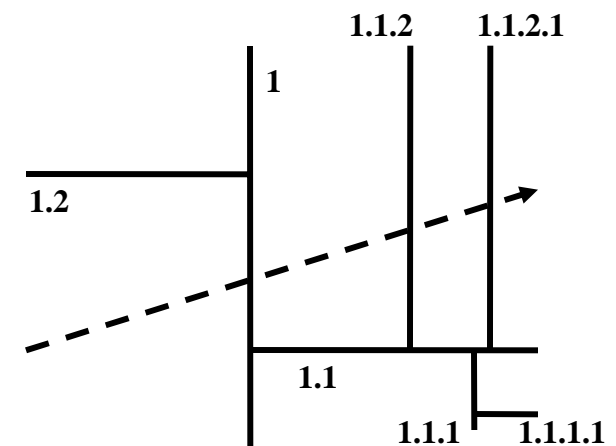
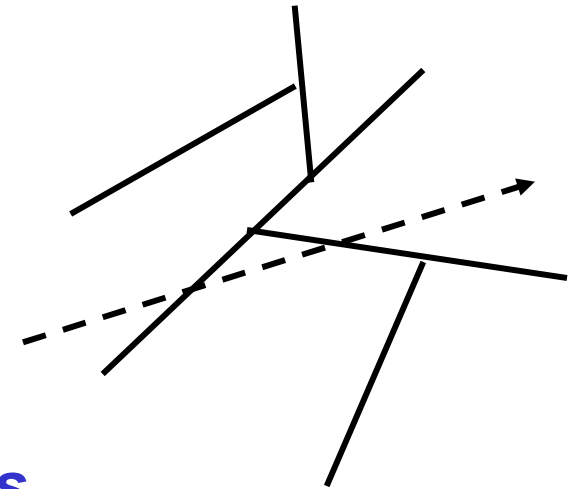
---

```
for each sample i
  for each sample j
    calculate the distance  $D[j] = d(i, j)$ 
  sort D
  output k first elements of D
```

- **Complexity  $O(N^2)$**
- **acceleration through the use of kD-Trees**

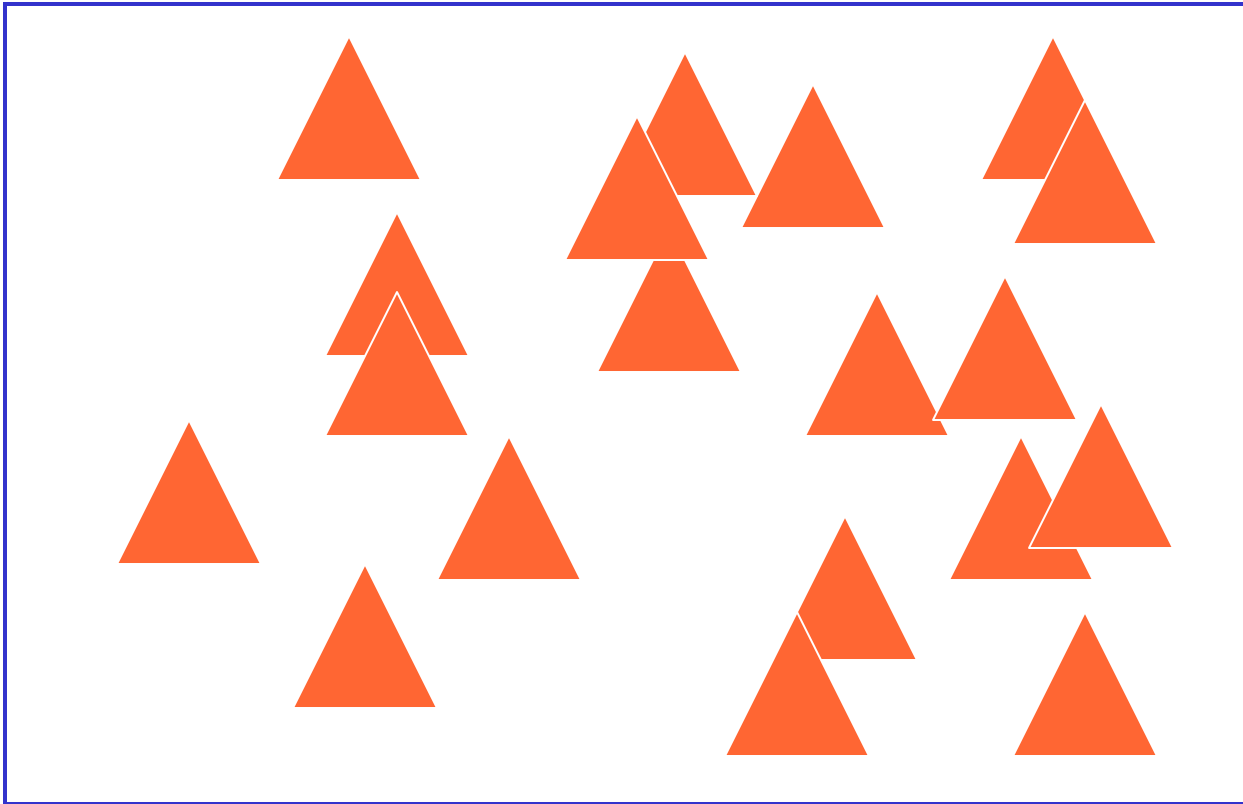
# BSP- and Kd-Trees

- **Recursive space partitioning with half-spaces**
- **Binary Space Partition (BSP):**
  - Recursively split space into halves
  - Splitting with half-spaces in arbitrary position
    - Often defined by existing polygons
- **Kd-Tree**
  - Special case of BSP
    - Splitting with *axis-aligned half-spaces*
  - Defined recursively through nodes with
    - Axis-flag
    - Split location (1D)
    - Child pointer(s)
  - See following slides for details



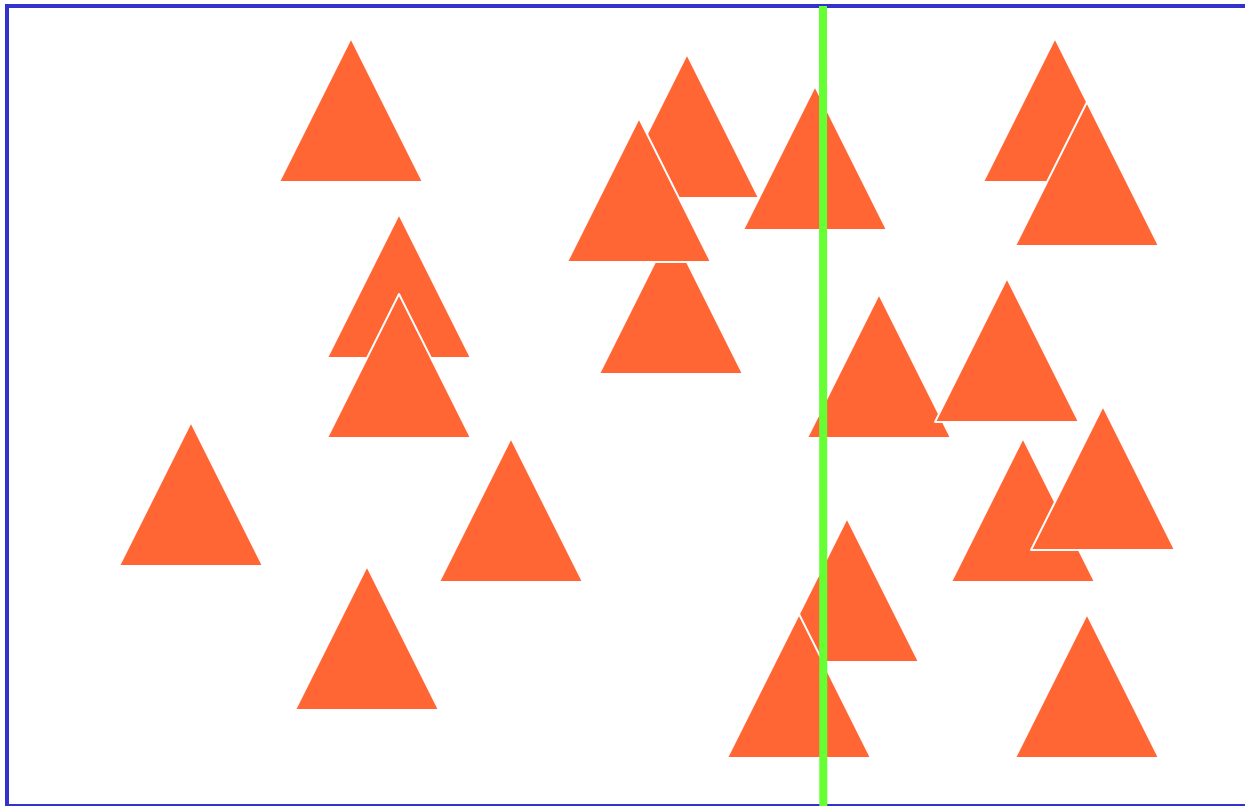
# kD-Trees

---



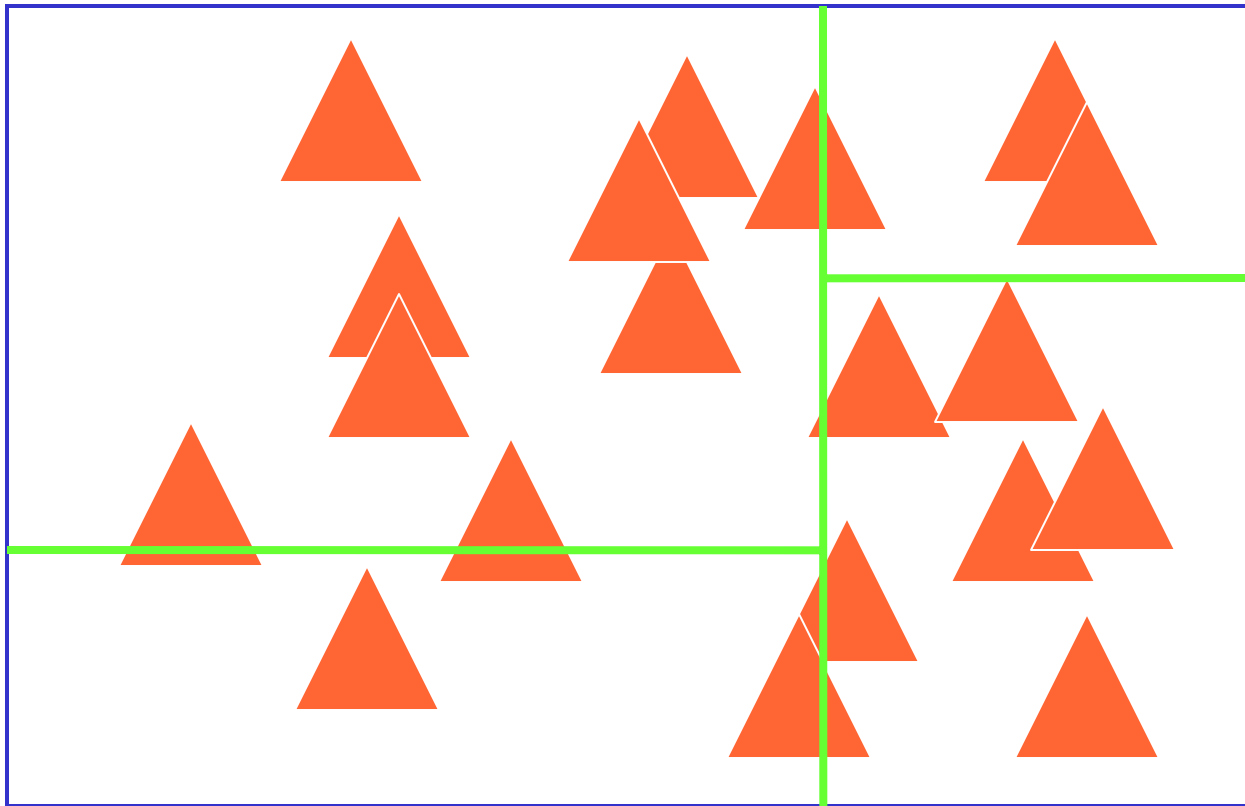
# kD-Trees

---



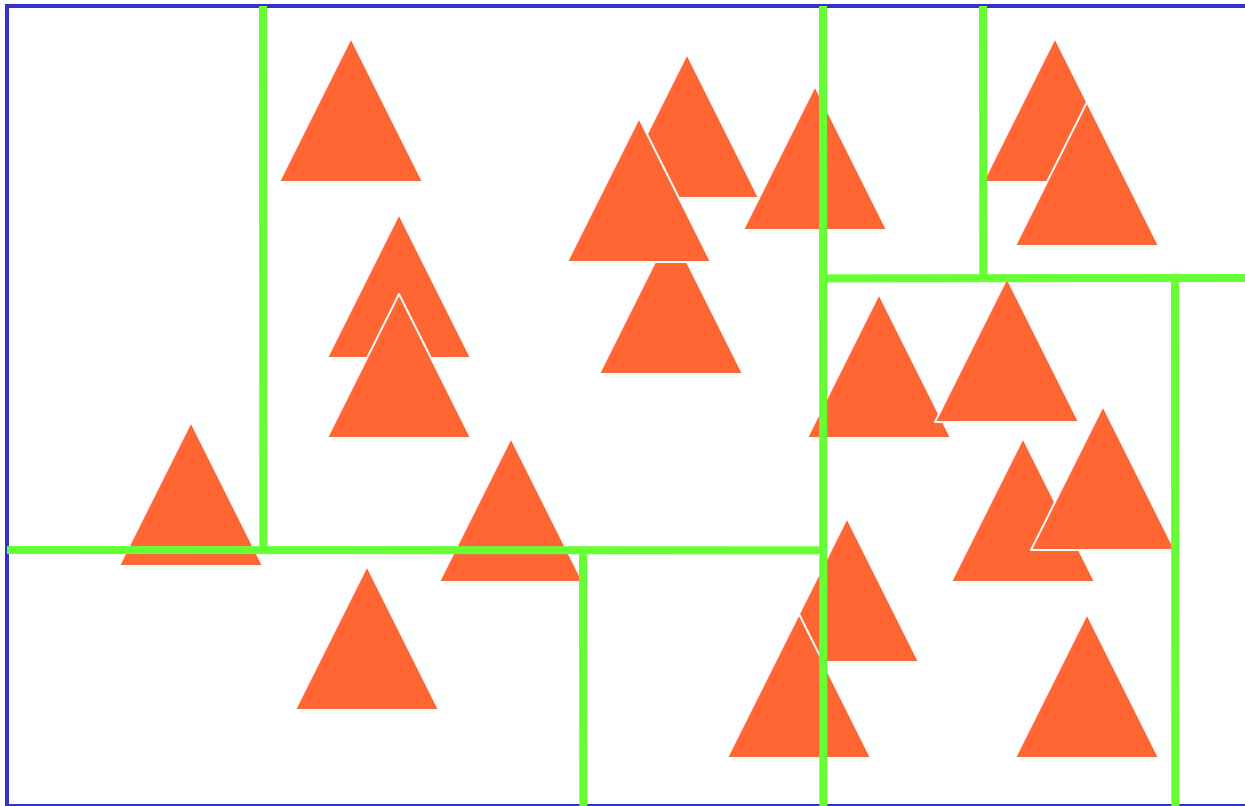
# kD-Trees

---



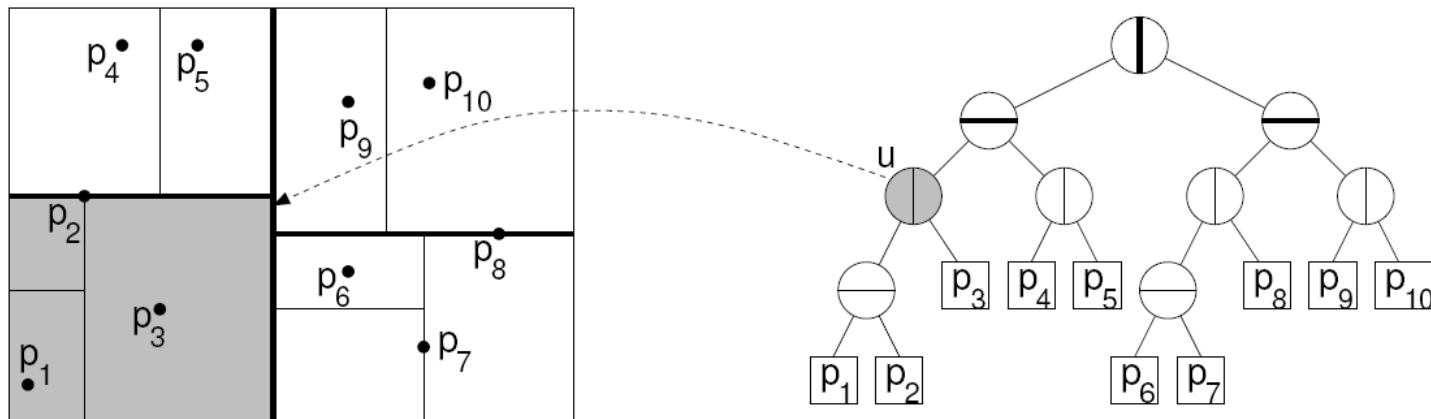
# kD-Trees

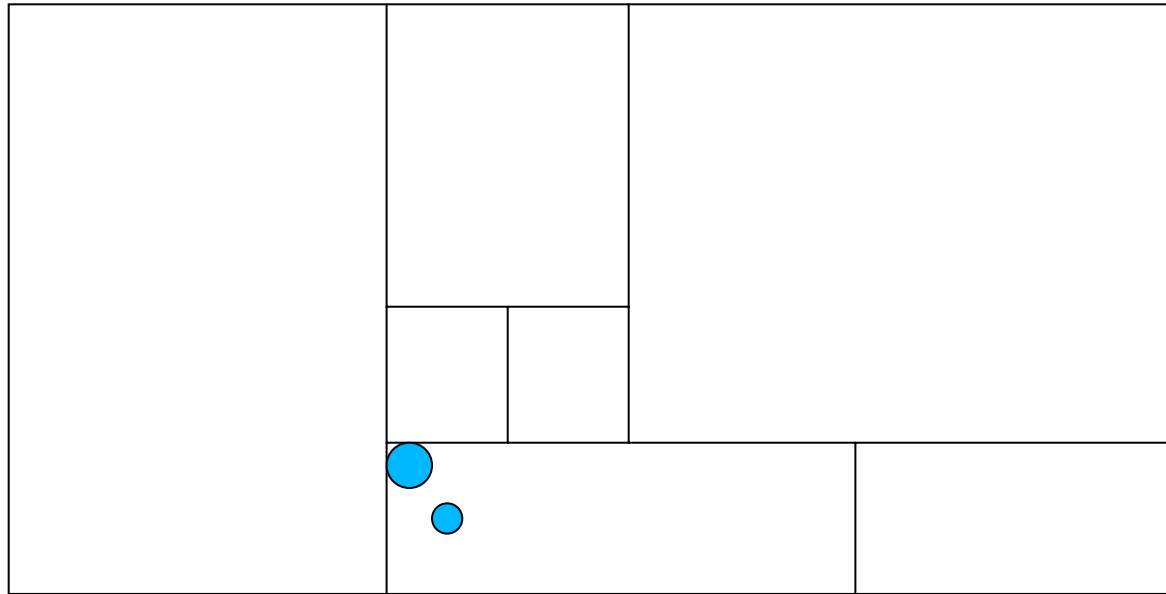
---



# ANN - Library

- See <http://www.cs.umd.edu/~mount/ANN/>
- Two different traversal modes
  - depth-first search
  - priority search





# Depth-first Search ANN

---

- **descent the tree**
- **process closer child first**
  - based on bounding box
  - incremental distance updates
- **let minDist be the minimum distance found so far**
- **when back process other child if**  
 **$\text{dist}(\text{child2}) < 1/(1-\epsilon) * \text{minDist}$**

# Priority-Search ANN

---

- **evaluate the distance for each child**
- **sort children in to priority queue**
- **process first element in priority queue if**  
 **$\text{dist}(\text{front}) < 1/(1-\epsilon) * \text{minDist}$**

# kNN in Cuda

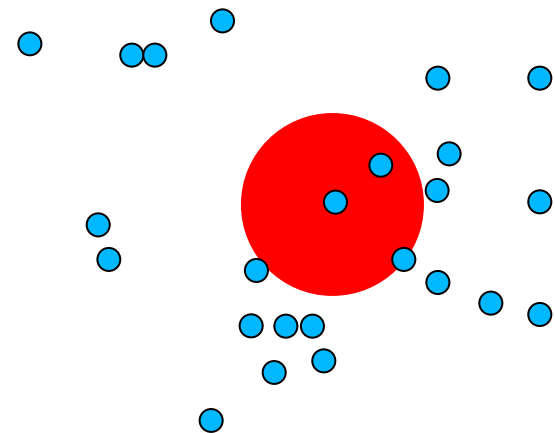
---

- **[Fast k Nearest Neighbor Search using GPU, Garcia & Debreuve, CVGPU 2008]**
- **The implemented only the brute force kNN method**
- **Speedup of 10-20 compared to ANN on CPU!**
- **How much faster can you get with a parallel data structure?**

# Bounded Nearest Neighbor Search

---

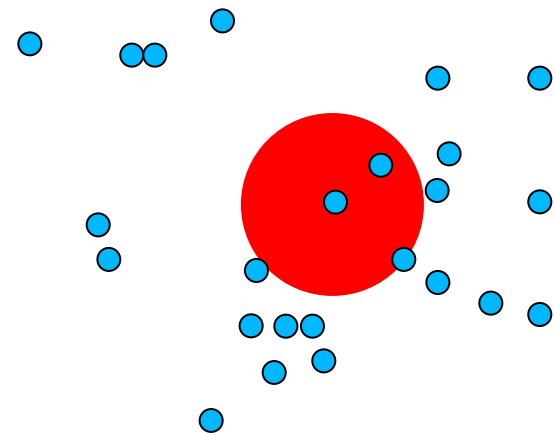
- **(Compare cell coverage exercise)**
- **Task: Search for all neighbors in a given radius  $r$  and perform some computation**



# Bounded Nearest Neighbor Search

---

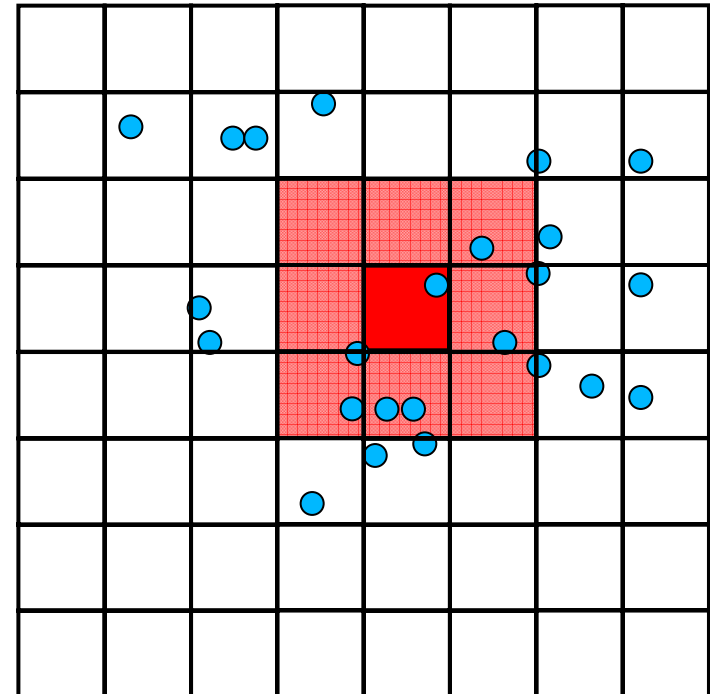
- **(Compare cell coverage exercise)**
- **Task: Search for all neighbors in a given radius  $r$  and perform some computation**



# Bounded Nearest Neighbor Search

---

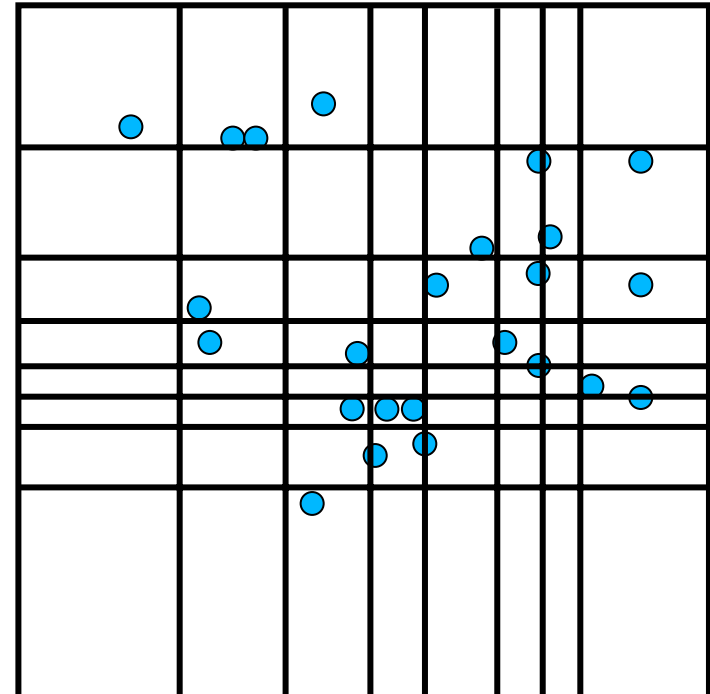
- **(Compare cell coverage exercise)**
- **Task: Search for all neighbors in a given radius  $r$  and perform some computation**
- **Sort into grid of radius  $r$**
- **Visit all neighboring cells**
- **Problem: load balancing**



# Using Tensor Product Grids

---

- **Task: Find only some nearest neighbors**
- **Adapt cell boundaries to occupation**
  - Can be done during the sorting  
(bucket sort with adaptive boundaries – pivot elements)
- **Might require to look in a larger neighborhood of cells**



# Summary

---

- **Parallel Design Patterns**
  - Decomposition
  - Dependence Analysis
  - Design Evaluation
- **Synchronization**
  - Beware of read-after-write and write-after-read hazards
  - Atomic Functions
- **Sorting**
  - Bitonic
  - Bucket, Radix, Hybrid -- all rely on PrefixSum/Scan algorithm