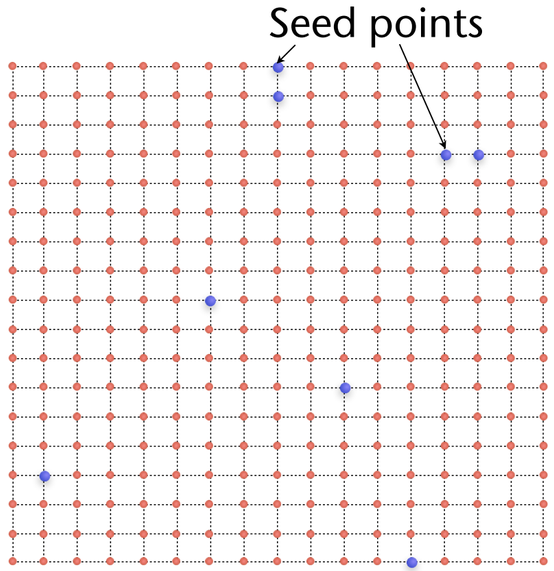
Massively Parallel Algorithms
Geometric Algorithms
(for now, only Distance Transform)

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

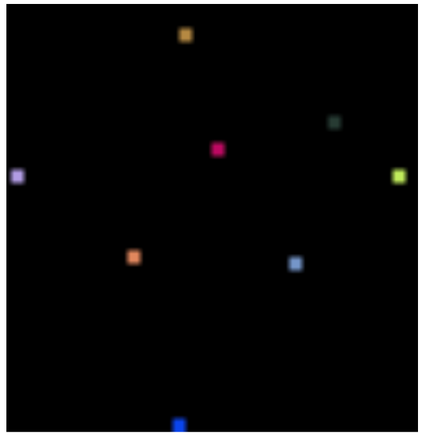
The Distance Transform



- Given a number of seed points: $S = \{\mathbf{s}_1,...,\mathbf{s}_k\}$ in 2D
  (higher dim. work similarly)
- The (Euclidean) distance transform (DT/EDT) stores for every point **p** the smallest distance to all seed points, i.e.,

$$\mathcal{D}(\mathbf{p}) = \min_{\mathbf{s}_i \in S} \{||\mathbf{p} - \mathbf{s}||\}$$

- Often, this is done on a grid
  (e.g., an image or 3D grid)
- Definition: Voronoi region of a seed point **s**
  = region of all nodes in space/the grid
  that are closest to this **s**
  (closer to **s** than to any other seed point)

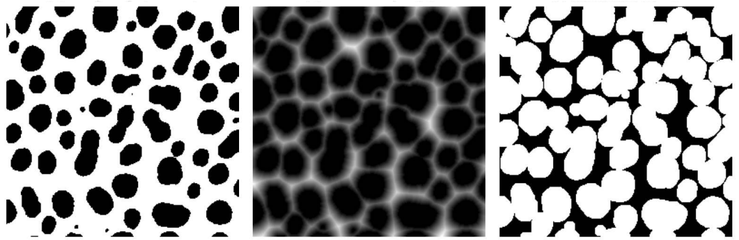


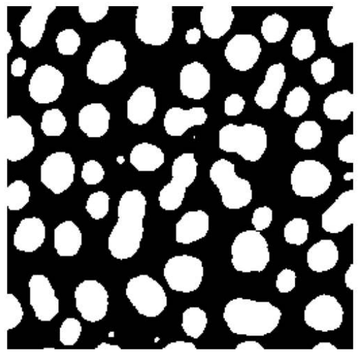
Seed points
(color = ID)

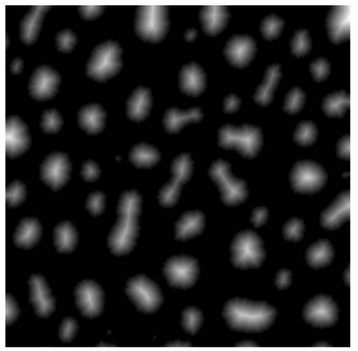
Nodes store the coords/ID
of their closest seed point

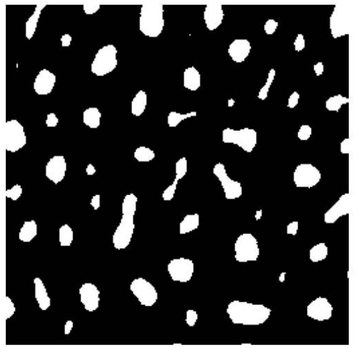
Simple Application: Morphological Image Transforms

- Given an input image (e.g., microscopy of soup of bacteria)
  - Usually B/W, white blobs = bacteria
- Remove noise, so we can count the real specimens
- Method: "erosion" operator
  - Compute DT, where black pixels = seed nodes
  - Threshold image, i.e., pixels with $\mathcal{D} > \tau$ are white, rest is black
- BTW: the "dilation" operator is just the dual
  1. Invert image,
  2. compute DT,
  3. threshold



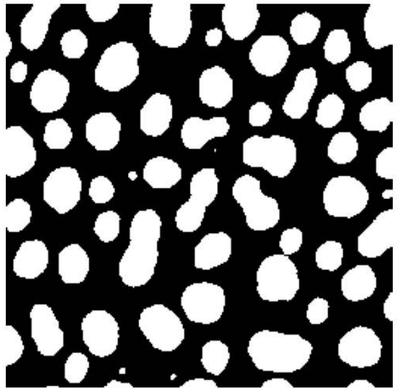


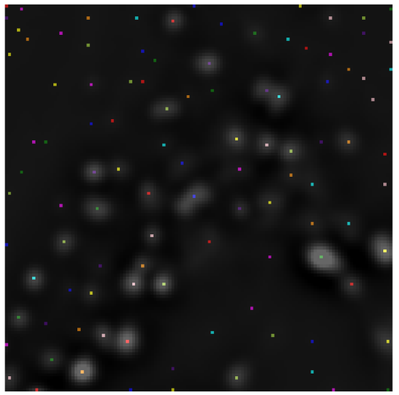


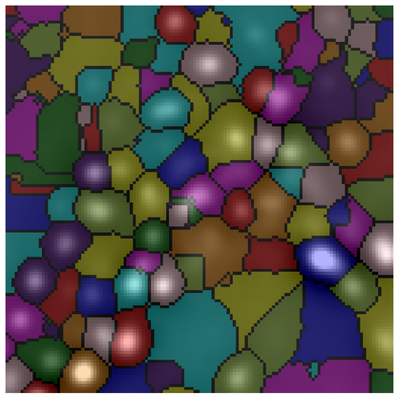


Object Segmentation

- Given BW input image (e.g., bacteria from microscopy); segment into "blobs" such that each one represents one specimen, but don't shrink them
- Method (simplified "watershed" transform):
  - Erosion operator (parameter τ differentiates between noise and real specimens)
  - Centers of specimen = barycenter of each blob (= average of white pixel coords of the blob)
  - Compute distance transform with those centers as seed points; here, we are only interested in the Voronoi regions
  - Each region with same seed (color) = "home" of one specimen
  - Overlay orig. BW image with DT; color white pixels with their corresponding seed color







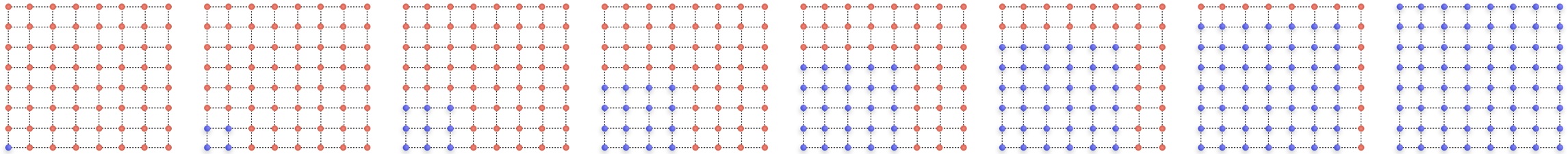
Simple Special Effects in Games

Flood Filling

- Data structure of the DT:
  - Regular 2D (3D) array, elements DT[*k*,*l*] store (*i*,*j*) = coords (array indices) of closest seed point
    - Usually also store the distance $\mathcal{D}$ to closest seed point (could always be computed on demand)
  - Seed nodes = elements in array with distance 0
- Flood filling = breadth-first walk through grid
  - Maintain p-queue storing nodes at the "edge" of the regions visited so far
    - Sort p-queue by distance (= distance from their respective closest seed point)
  - Put each seed node in p-queue, with distance 0
  - Initialize all non-seed nodes with distance = +∞
  - Let **p** = front of p-queue:
    - Let **s** = closest seed point of **p**
    - Consider all $\mathbf{q}_i$ in 8-neighborhood around **p**: if dist($\mathbf{q}_i$,**s**) < $\mathcal{D}(\mathbf{q}_i)$, then store **s** ⟶ DT[$\mathbf{q}_i$] (so far), and put $\mathbf{q}_i$ into p-queue

Very Inefficient Parallel Version of Flood Filling

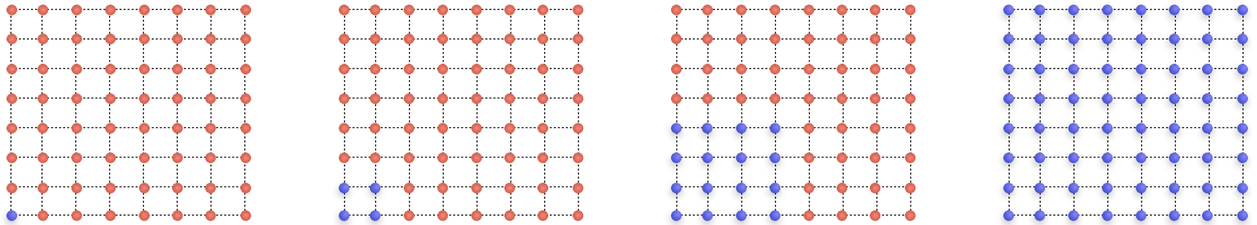
- One thread per node
- Each node **p** becomes "active" whenever it receives a new closest seed point **s**
  - In that case, distribute **s** to neighbors **q**, *provided* dist(**q**,**s**) < $\mathcal{D}(\mathbf{q})$
  - Use two DT's in ping-pong fashion for each iteration
- Maximum # iterations = n , where $n^2 = N$ = #nodes/pixels



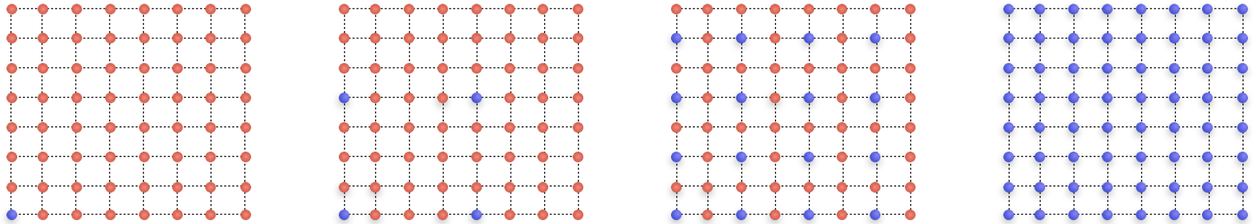
The Jump Flooding Algorithm (JFA)

[2006]

- Idea 1: double the offset d for "neighborhood" in each round, i.e., node **p** distributes its seed to nodes $\mathbf{q} = (p_x \pm d, p_y \pm d)$, $d = 1, 2, 4, 8, \ldots$



- Idea 2: start with $d = n/2$, then halve the offset in every round



  - Theoretically, results should be equal, experience shows that second version is better

- Idea 3: turn *scatter* operation into *gather* operation
  - In previous version, each node gets overwritten with a seed from (at most) 8 neighbors at offsets (±*d*,±*d*) (use atomic op's!)
  - Equivalently, each node $\mathbf{q}$ gathers seeds of all 8 neighbors $\mathbf{p}_i$, then stores

$$\mathcal{D}(\mathbf{q}) = \min\{\mathcal{D}(\mathbf{q}), \mathcal{D}(\mathbf{p}_1), \ldots, \mathcal{D}(\mathbf{p}_8)\}$$

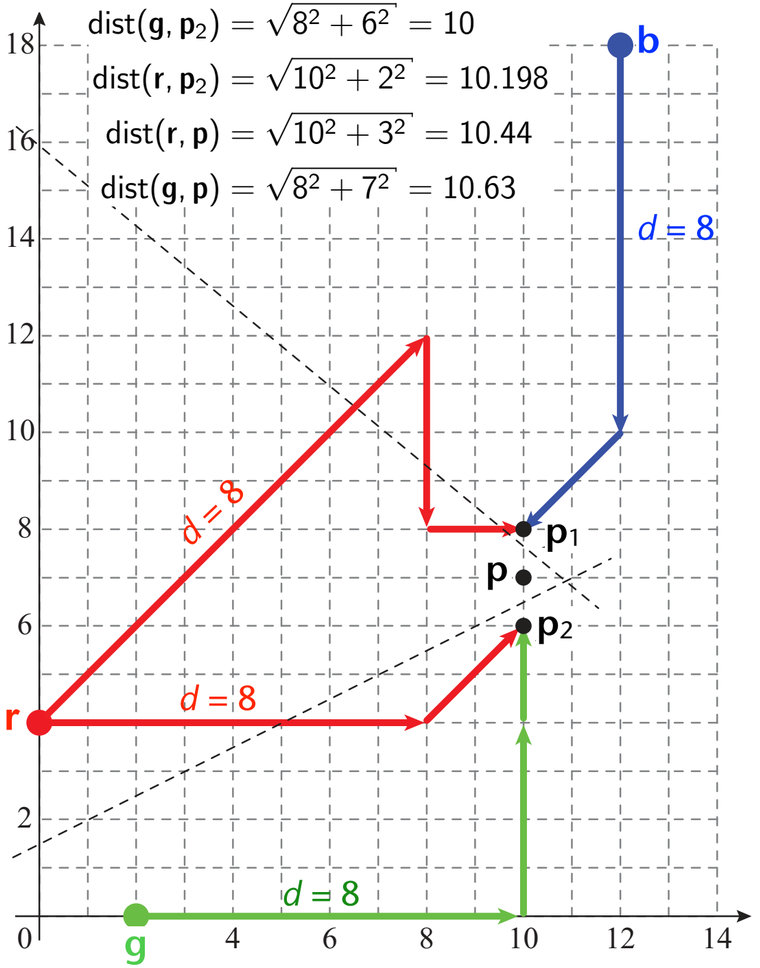
    (with corresponding seed)
  - No atomic operation needed any more, just ping-pong DT arrays

Complexity

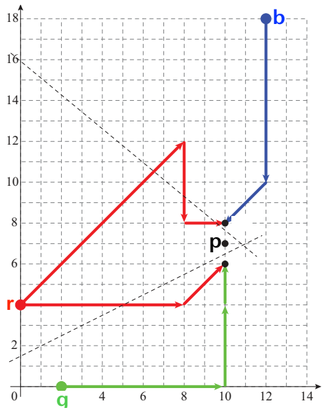
- Work complexity of (serial) flood filling: O(N)
  - Each node gets visited exactly once, work per node is constant
- The inefficient parallel algo:
  - Depth complexity = $O(n)$
  - Work complexity = $O(n \cdot N) = O(N^{1.5})$
- The Jump Flooding Algorithm:
  - Depth complexity: $O(\log n)$
  - Work complexity: $O(N \cdot \log n) = O(N \cdot \log N)$

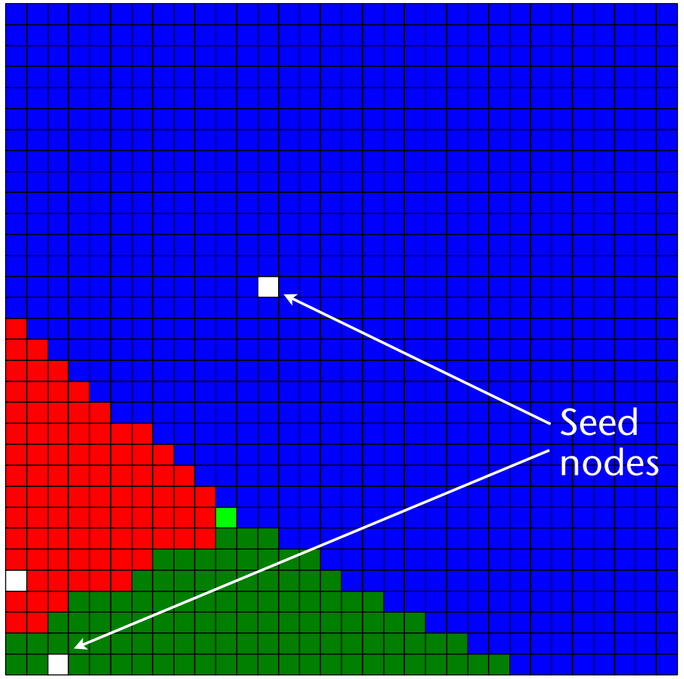
Error Analysis

- JFA fails to propagate the correct seed point in rare cases!
  - I.e., the DT computed by JFA is only *approximate*
  - Depending on the appl., this might not be a problem - especially, if the $\mathcal{D}$ values are close
- Observation: if a seed **s** reaches a node **p**, it was propagated there through a number of other node points $\mathbf{p}_i$ ⟶ path from **s** to **p**
- Consider this example: three seeds **r**, **g**, **b**, and grid point **p**
  - The path from **r** to **p** must pass through $\mathbf{p}_1$ or $\mathbf{p}_2$ (can you explain, why?)
  - Nodes $\mathbf{p}_1$ and $\mathbf{p}_2$ will not take **r** as seed, because **g** and **b** are closer, resp.



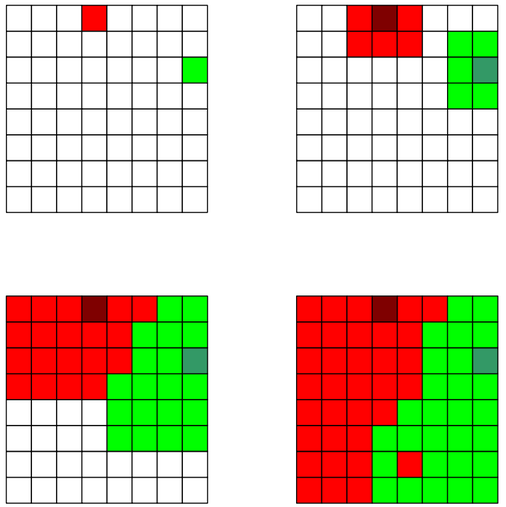
- Result:
  pixel centers = grid nodes





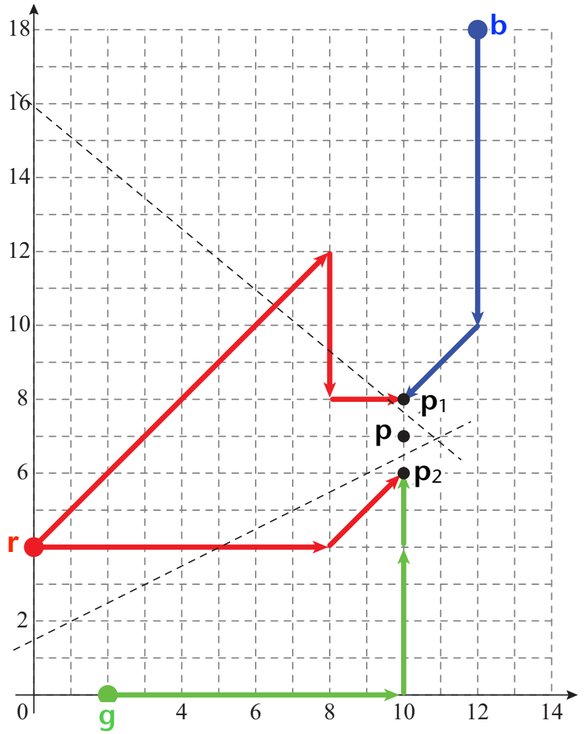
- It is possible to quantify the error probability for certain grid nodes

- Sufficient condition for **p** to be correct:
  For a grid node **p** to receive its closest seed **s**, there must exist at least one path from **s** to **p**, such that *each node on the path* has **s** as its closest seed!
  (Even if some nodes on the path get overwritten later.)
- Intuitively, this is the reason why the JFA variant using "doubling offset size" creates more erroneous nodes than the "halving offset size":
  - Nodes close to seeds get the correct seed in early iterations
  - They will never change their seed later
  - Other seeds cannot "travel" through such already-determined regions



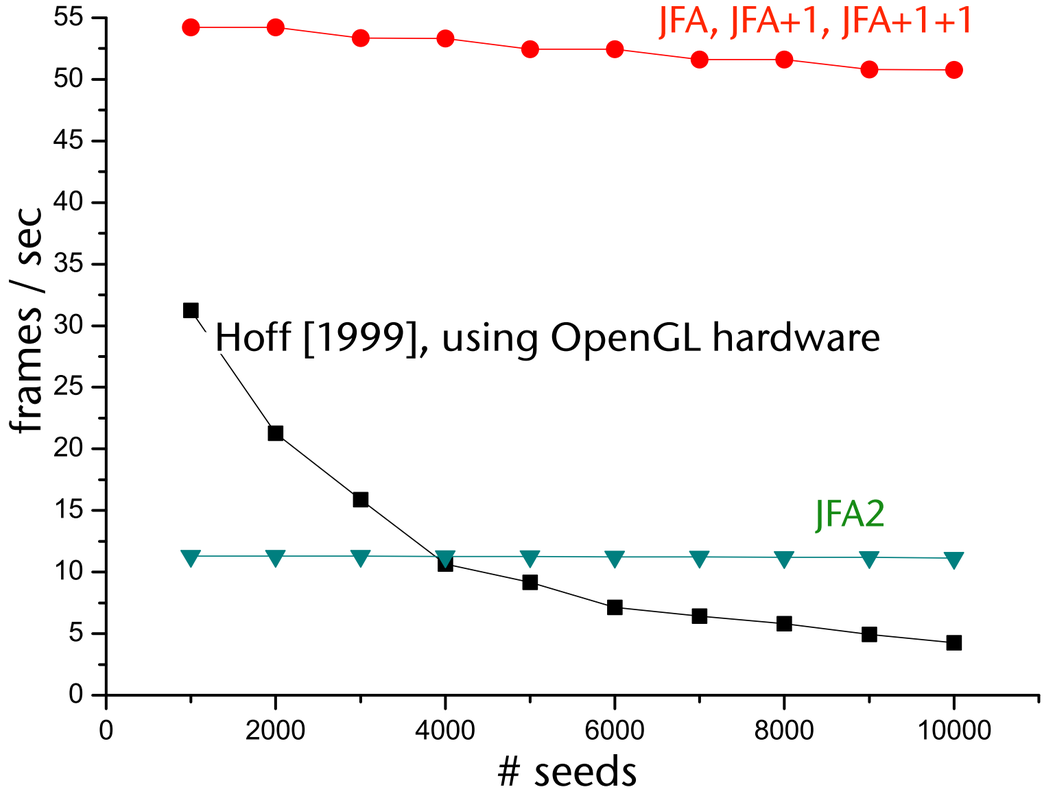
Variants of JFA

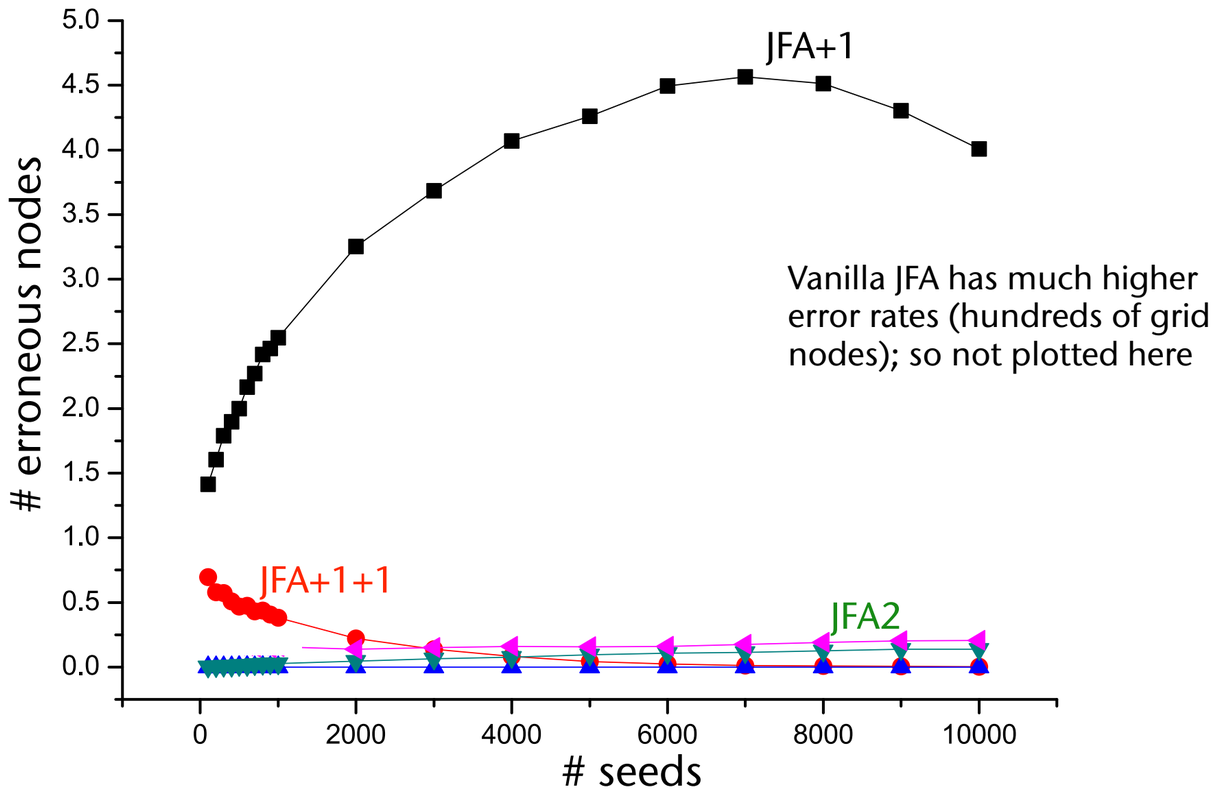
- JFA+1: after the orig. JFA, execute one more iteration with offset size d=1
  - One could extend this to JFA+1+1, etc.
- JFA2: store *two* seeds per grid node
  - Motivation: in the failure example, r is indeed the second closest seed to $\mathbf{p}_1$ and $\mathbf{p}_2$!
  - Therefore, always propagate the two closest seeds along the paths
  - Consequence: 2x the computational effort (each node needs to compute the min over 18 values, instead of 9, in each iteration)



- JFA randomized:
  - Store all seeds in an (additional) table
  - If a node does *not* receive any seed, pick a random seed from the table

Performance

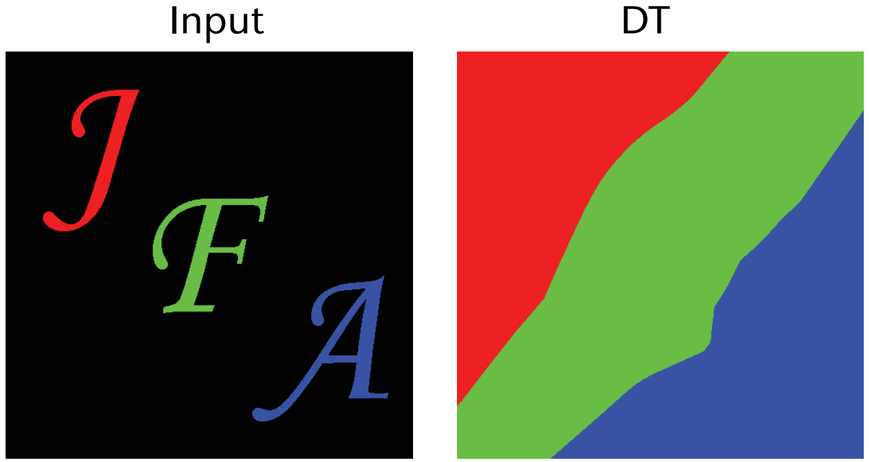




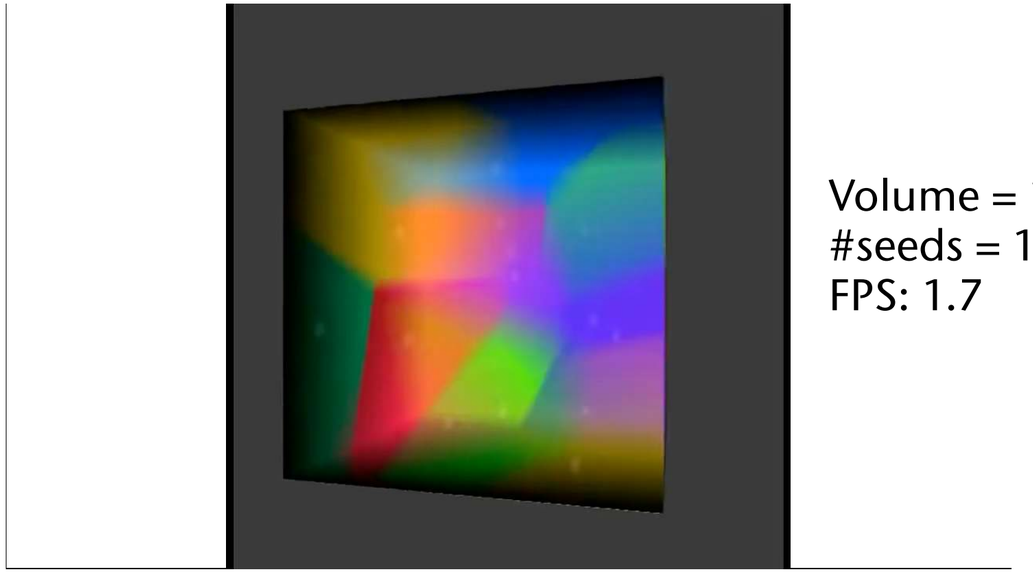
Grid size 512x512, GeForce 6800 [2006]

Remarks

- Grid-based (approximate) DT's work for any kind of input



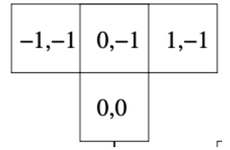
- The algos work just the same in higher dimensions



Volume = 128^3,
#seeds = 15,
FPS: 1.7

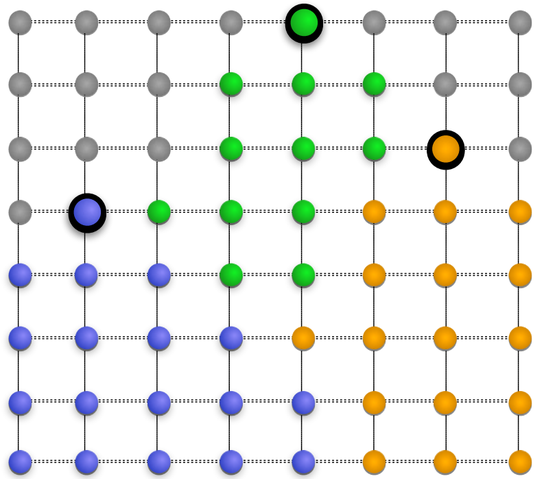
The Plane Sweep Propagation Algorithm

- Instead of a breadth-first walk, use a simple node-by-node propagation in *one* direction (which will, for the moment, produce an incomplete DT)
  - For downwards propagation, each node considers only its 3 neighbors above (and itself): $\mathcal{D}(\mathbf{q}) = \min\{\mathcal{D}(\mathbf{q}), \mathcal{D}(\mathbf{p}_1), \ldots, \mathcal{D}(\mathbf{p}_3)\}$



  - The algorithm for the *down sweep*:

```
parallel for all columns j:
  loop i = 1, .., n-1:
    if DT[i,j] = seed: continue      // do nothing
    DT[i,j] = closest_seed( DT[i,j],   DT[i-1,j-1],
                            DT[i-1,j], DT[i-1,j+1 )
    synchronize all threads
```



- Coalesced memory access: use *row-major* storage for down-/up-sweep, and *column-major* for left-/right-sweep!

Enter Cooperative Groups in CUDA

- Synchronizing the whole grid: use cooperative groups in CUDA = user-defined groups of threads of "any" size
- Example (mix of C++ and pseudo code):

```
namespace cg = cooperative_groups;
__global__ void downSweep( DT )
{
  cg::grid_group grid = cg::this_grid();
  j = threadIdx.x + blockDim.x*blockIdx.x;
  for i = 1, .., n-1:
  {
    if DT[i,j] = seed:
      continue;                          // do nothing
    DT[i,j] = closest( DT[i,j], DT[i-1,j-1], DT[i-1,j], DT[i-1,j+1 );
    grid.sync();
  }
```

Must use cudaLaunchCooperativeKernel()!

- Observations on the different synchronization methods (2020):
  - The performance of block synchronization is related to the number of warps involved, and the performance of grid level synchronization is mainly affected by the number of blocks involved. [Zhang et al., 2020]
  - The performance of grid sync is acceptable if #blocks/SM ≤ 2
- Caveats on **`grid.sync`**:
  - The GPU cannot swap out waiting blocks for new ones!
  - It cannot hide global memory latency any more!
  - The GPU's hardware only natively supports intra-block barriers
  - Grid synchronization is implemented via complex hardware-accelerated semaphores and tree barriers

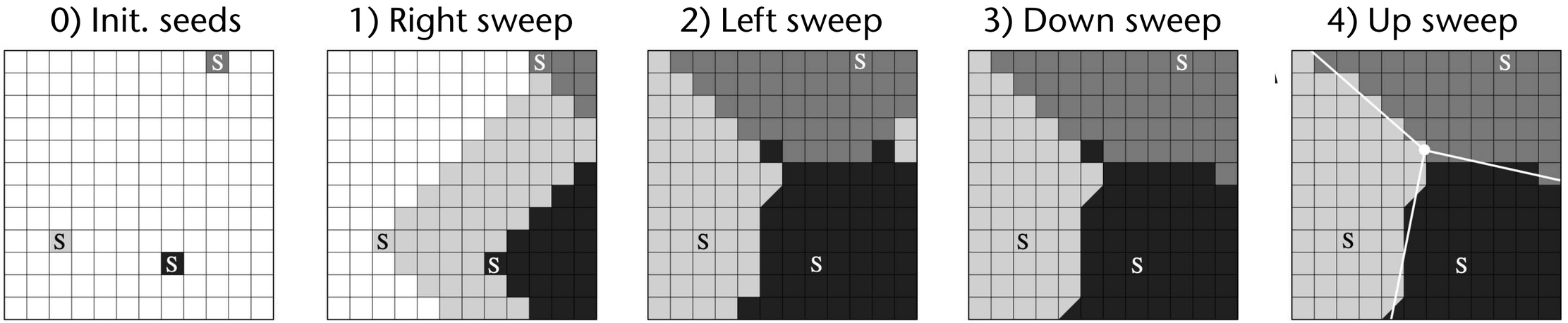
Overall Plane Sweep Algorithm

- Similar to *down sweep*, define *up* sweep, *left* sweep, and *right* sweep
- The algorithm as a whole:
  - Perform the four sweeps, each one generates a **`DT_dir`**, where **`dir`** is one of "left", "right", "up", "down"
  - For all nodes **p** in parallel, compute

$$\mathcal{D}(\mathbf{p}) = \min\{\mathcal{D}_{\text{left}}(\mathbf{p}), \mathcal{D}_{\text{right}}(\mathbf{p}), \mathcal{D}_{\text{up}}(\mathbf{p}), \mathcal{D}_{\text{down}}(\mathbf{p})\}$$

- Practical questions:
  - Can you use warp-level shuffling primitives (?)
  - Can you do all 4 sweeps in parallel, then merge the results using $O(N)$ threads?
  - Best way to partition the work in the last step: 1 block = 1 row of nodes, or 1 column of nodes, or a block of nodes?

- Visualization of an example (each sweep is merged with the previous ones):

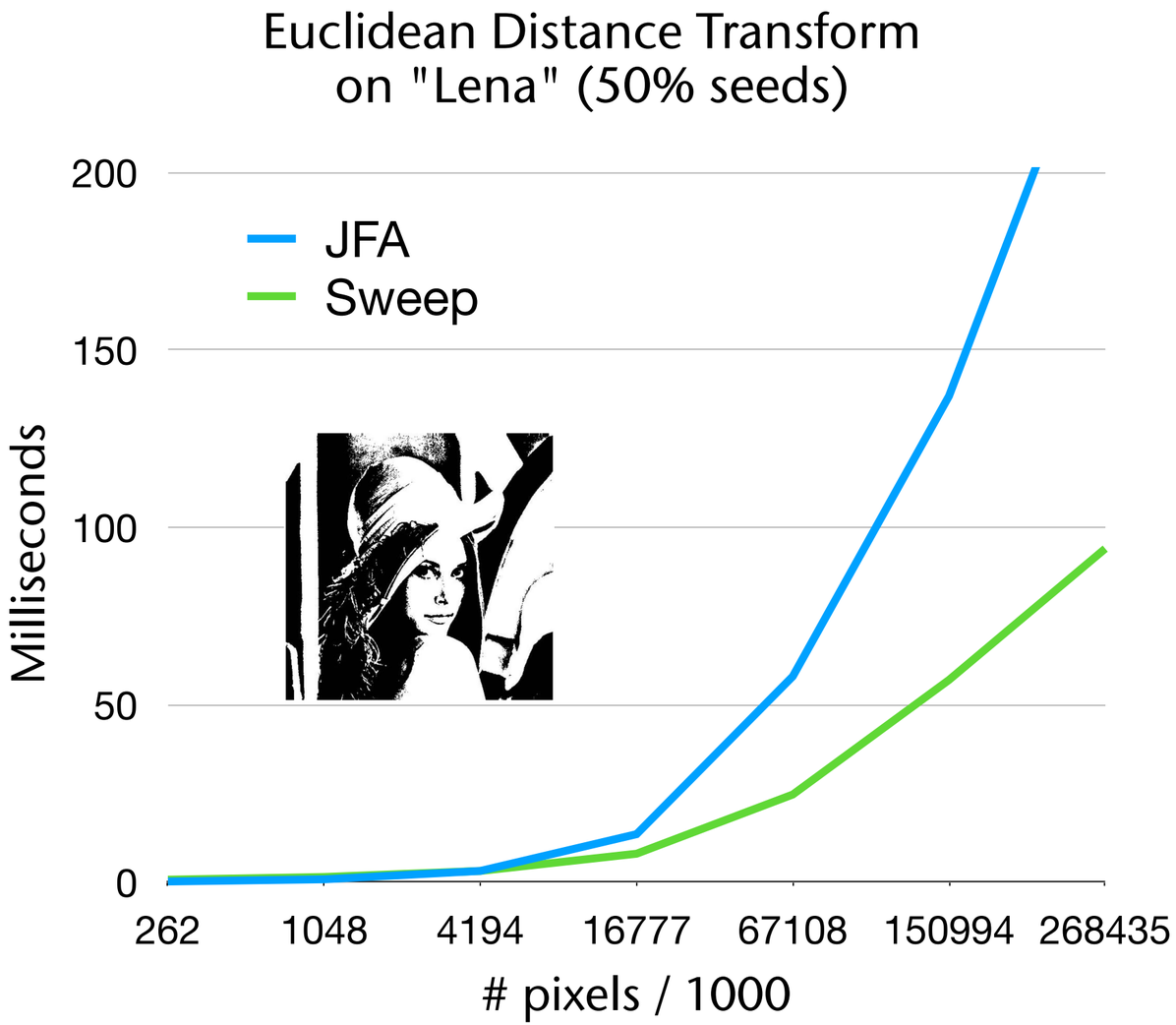


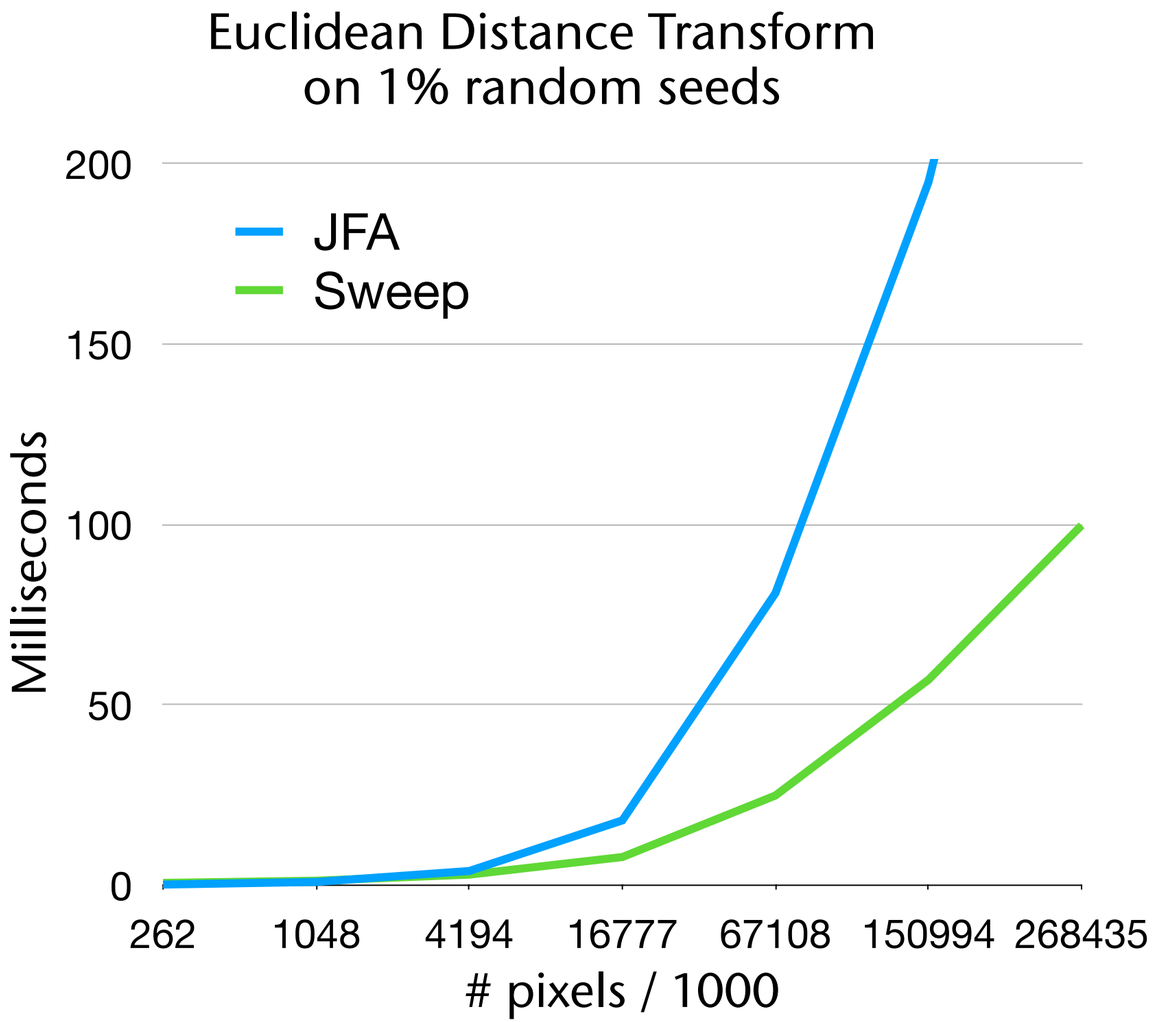
- Extension to higher dimension *d*: perform *2d* many sweeps
- Question: what if we perform all sweeps in parallel → *2d* "one-way DT's", then merge them using another kernel (one thread per node)?
  - Correctness? Performance?

Performance

- Complexity:
  - One sweep: depth complexity is in $O(n) = O(\sqrt{N})$, using $O(\sqrt{N})$ threads
  - In total: $O(\sqrt{N})$ depth complexity, $O(N)$ work complexity
- Optimality: the 4-way sweep *is* work-efficient, jump flood fill *is not*
- Here, work complexity also translates into memory bandwidth (both algos)!
- OTOH, JFA can utilize more parallelism, especially in the last rounds

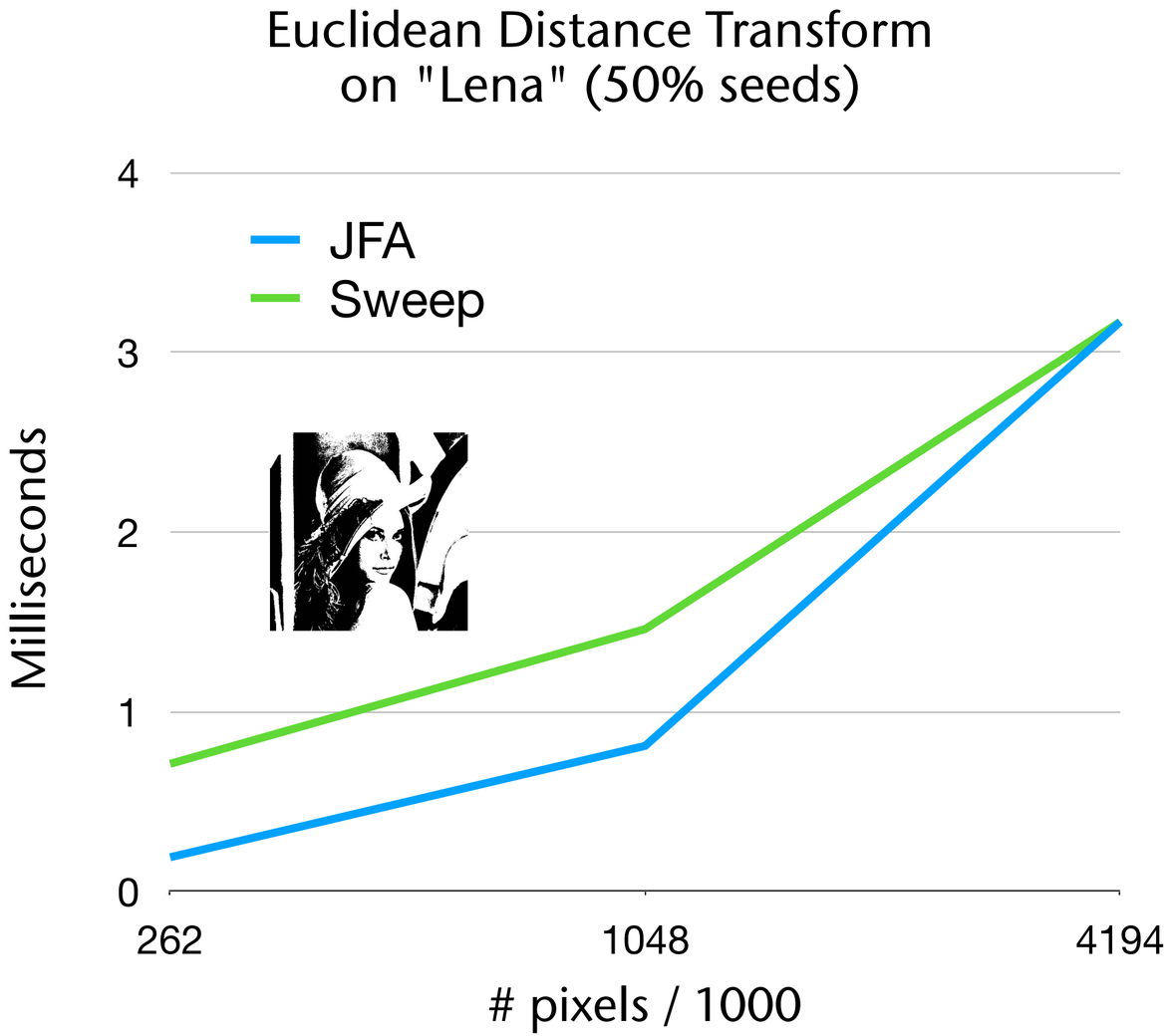
Performance Comparison

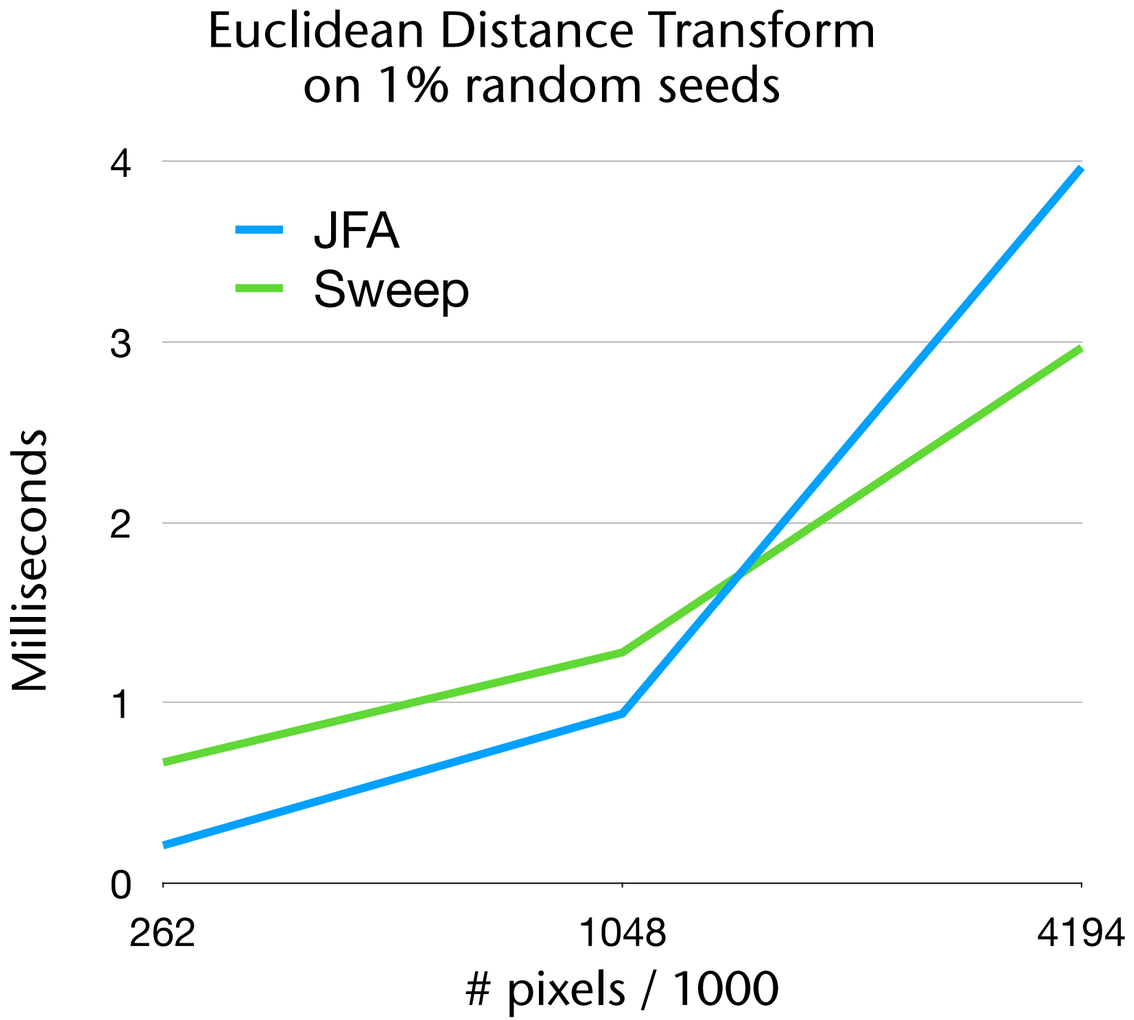




[Honda et al, 2017. Platform: GeForce GTX 1080]

Performance on Small Inputs

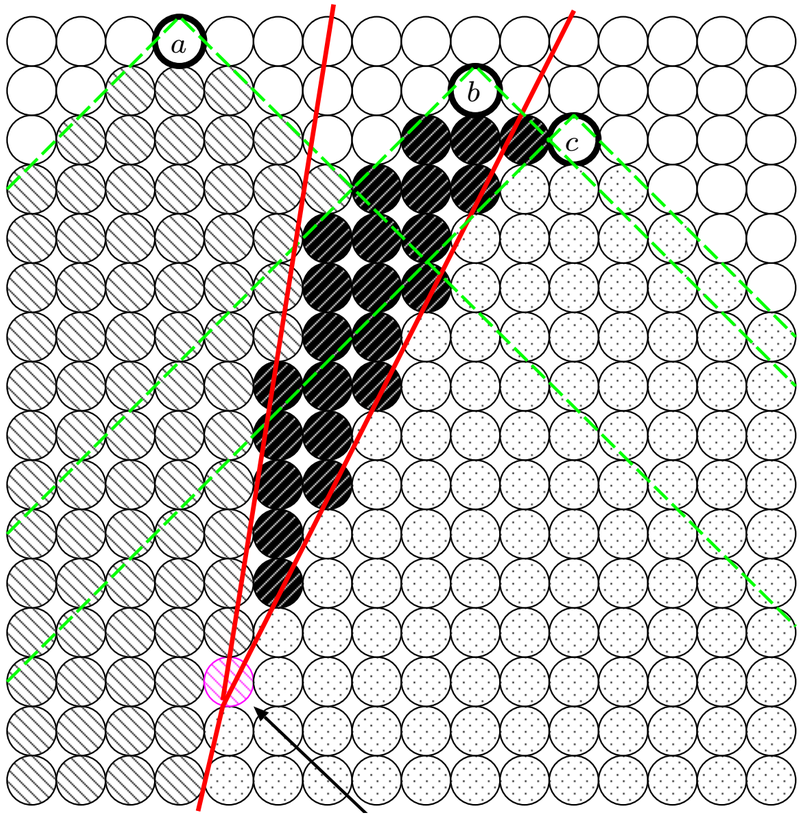




[Honda et al, 2017. Platform: GeForce GTX 1080]

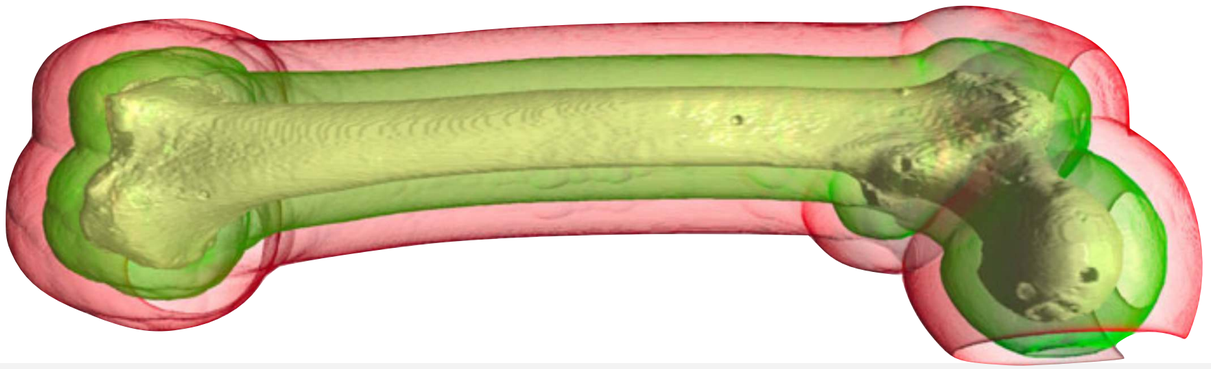
Error Analysis

- A node **p** keeps a wrong seed, if **p** cannot be "reached" by the correct seed
- This can happen only, if *all* nodes in **p**'s direct 8-neighboorhood have different closest seeds
- This means, node **p** must be contained in the Voronoi region of its closest seed (obviously), but none of its neighbors are in the same Voronoi region!
  ⟶ "exclave pixel"
- Example: after the down sweep, the pink node should have seed "b" as closest seed



More Applications

- Offset surfaces:
  - Given mesh, distance *d*
  - Compute offset surface = mesh with distance *d* from original mesh
  - Notoriously difficult to compute, because of potential undercuts
  - Simple approximation: convert mesh into 3D grid, nodes are inside or outside
  - Compute distance transform on grid
  - Nodes with distance < *d* are marked as "inside"
  - Convert back to mesh (see Advanced Computer Graphics: marching cubes)



- Accelerating ray casting (especially volume data sets):
  - Given objects, compute distance transform (DT)
  - Assume a ray has entered a voxel (with 8 nodes at the corners)
  - Let *d* = min of the DT's of the 8 nodes
  - Then, the ray cannot hit any object within a ball of radius *d* around the 8 nodes
  - Use for "empty space skipping"
  - A.k.a. "sphere tracing"

- Path planning:
  - Abstraction: the "robot" is just a point moving in the plane
  - Given a set of obstacles, a start position, and a goal position
  - Sample space by a grid, obstacles are seed nodes, compute DT
  -