Massively Parallel Algorithms
Parallel Hashing & Applications

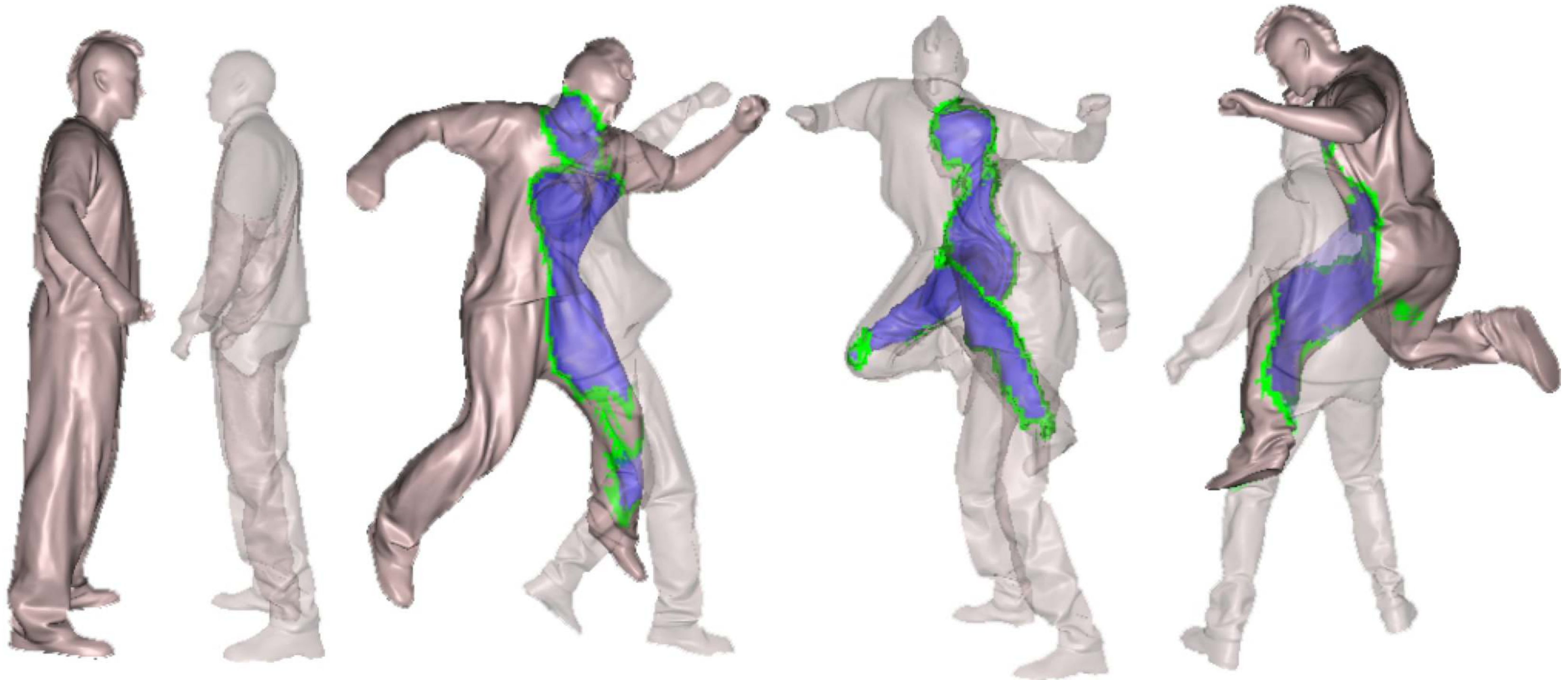
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

The Dictionary as an Abstract Data Type

- Frequently, the following operations are needed in an algorithm and executed a lot of times:
  - Insert (key,value)
    - Sometimes, keys are unique, sometimes not!
  - Retrieve a value by its key (or all values with the same key)
- Wanted: O(1) for both operations
- Implementations:
  - Hash table
  - Sorted array? nope, not even amortized complexity is in O(1)

Application: Intersection of Point Clouds

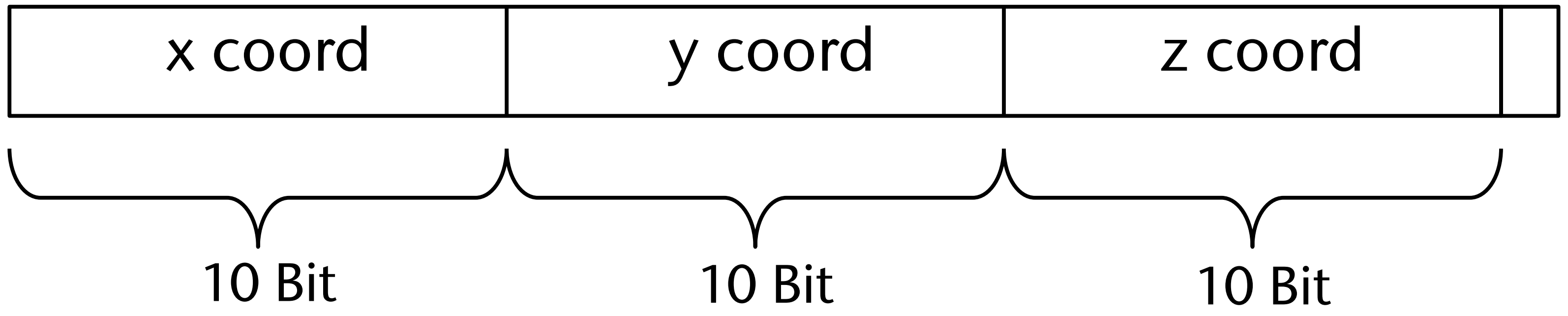
- Given: two point clouds representing two surfaces
- Task: compute "intersection" of the surfaces
  - If surfaces are continuous $\longrightarrow$ intersection is usually a set of curves in space
  - Here: intersection = set of points close to those curves
- Approach:
  - Superimpose background 3D grid
  - Find voxels occupied by both surfaces



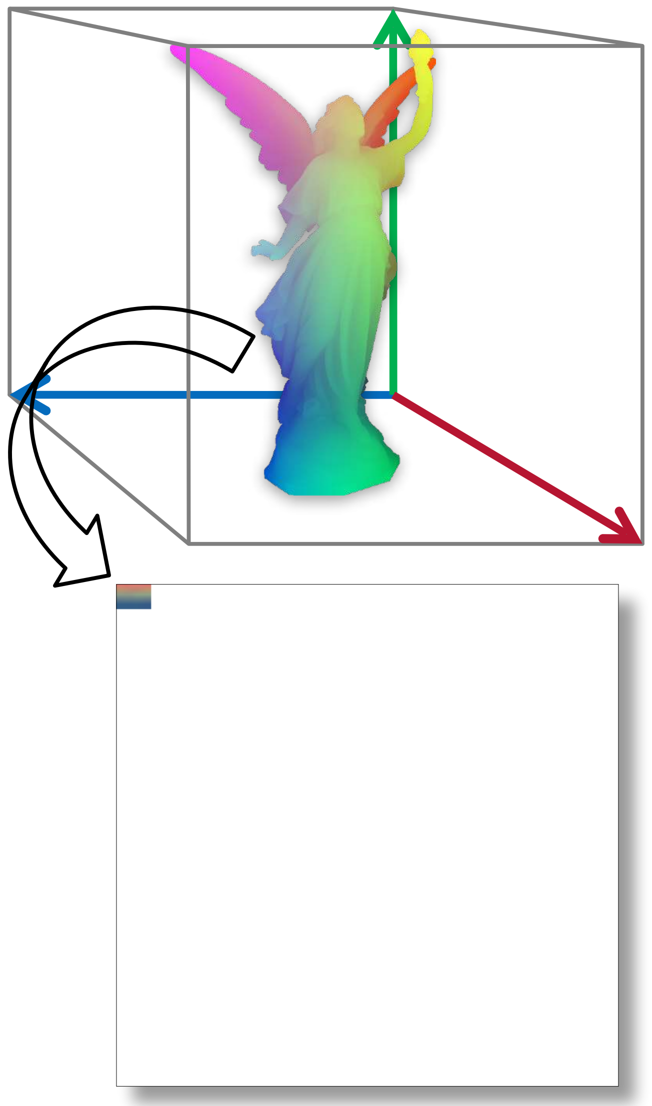
[Alcantara et al., Siggraph 2009]

Representing Geometry in a Voxel Grid

- Voxel grid = 3D grid partitioning space, voxels = empty or occupied
- Example:
  - 1024^3 voxel grid ≈ 1 billion voxels
  - Only 3.5 million voxels occupied ≈ 0.33%
- In practice: # occupied voxels ∈ $O(N^2)$, where N = voxel grid resolution
- Idea: store voxel grid in hash table (aka. spatial hash table)
  - Key = integer coordinates

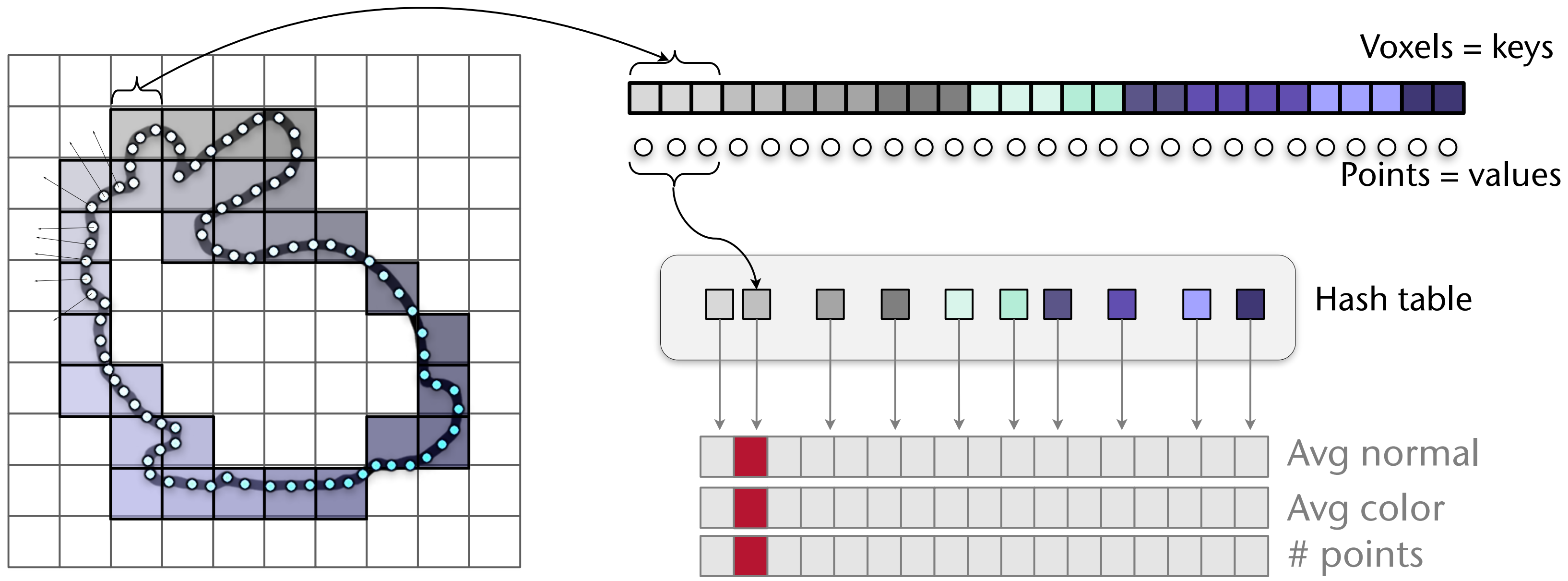


    - Or any other arrangement (e.g., Morton code)
  - Value = color, normal, …



Algorithm for Point Cloud Intersection

- Given: two point clouds *with normals*
  - E.g. from Kinect, upload to GPU
- First stage: build spatial hash table using one thread per point
  - Transform point by user-defined transformation (e.g., viewpoint transform)
  - Calculate integer *x*, *y*, *z* coordinates (scaling / rounding)
  - Assemble key (shift bits, or interleave bits for Morton code)

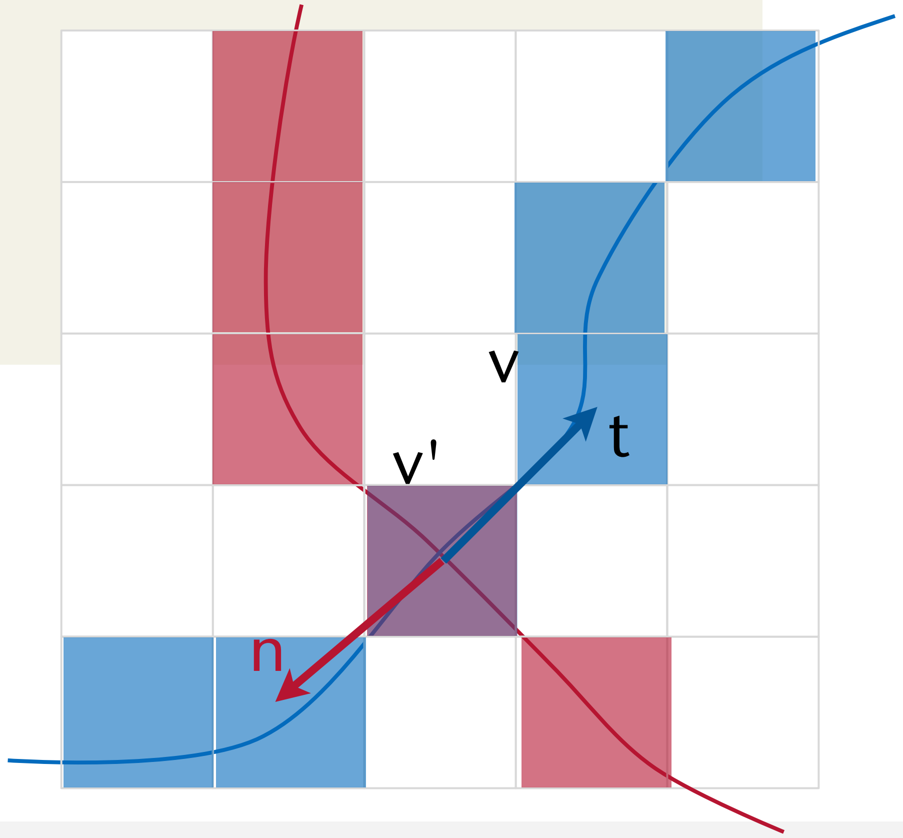


- Second stage: find intersecting voxels
    - One thread per occupied voxel
        - Translates to one thread per hash table slot, empty slots/threads do nothing

```
v = voxel of thread
v' = corresponding voxel in other object's hash table
if v' is occupied:
   mark both v and v' as intersecting
```

- Third stage: determine voxels inside/outside of surface
  - One thread per occupied voxel (for both objects in parallel)

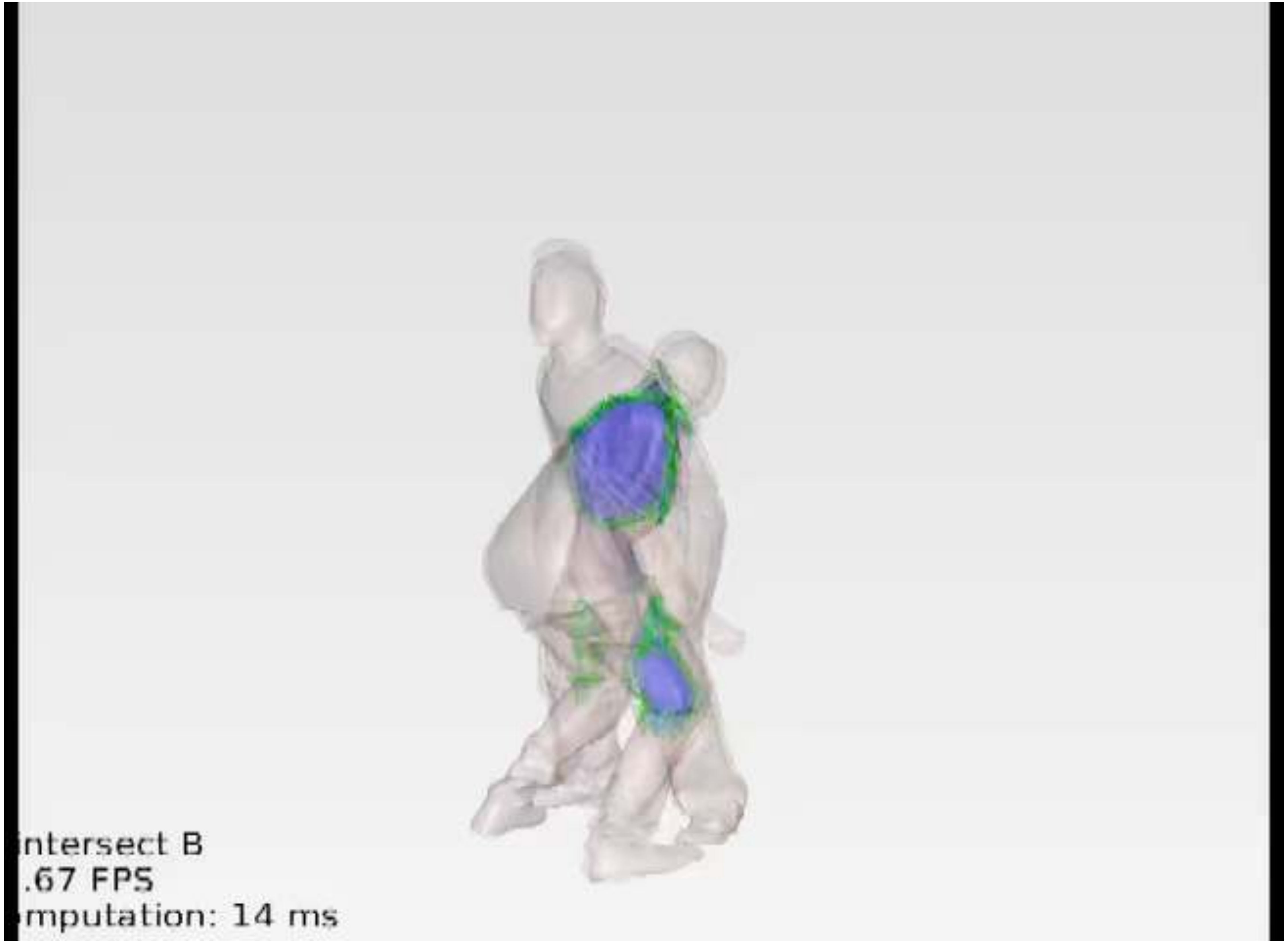
```
v = voxel of blue thread
if v not intersecting and
   v has intersecting neighbor v':
     t = v - v'     // a "tangent" to the blue surface in v'
     n = normal of voxel in red object corresponding to v'
     normalize n and t
     if t*n < cos(110°):
        mark v as "inside red"
     if t*n > cos(70°):
        mark v as "outside red"
```



- Fourth stage: propagate inside/outside status *along surface voxels*
  - One thread per occupied voxel
  - Do nothing, if own status is already set
  - Otherwise, repeatedly check neighboring voxels, copy their status, as soon as they've got one
  - Loop until **`__syncthreads_count`** or **`__syncthreads_or`** yields 0
    - Def. of **`int __syncthreads_count( int predicate )`**:
      like syncthreads, but evaluate predicate for all threads (in block), and return number of threads for which it is non-zero (each thread gets the same count)
    - Here, devise predicate that tells whether a thread has changed its status during current iteration
- Performance: ca. 20 msec/frame
  - Voxel grid = 128^3 , point cloud = 160k
  - Upload of point clouds takes another 5-10 msec / frame
- Also possible: Boolean operations on the surfaces

Bremen

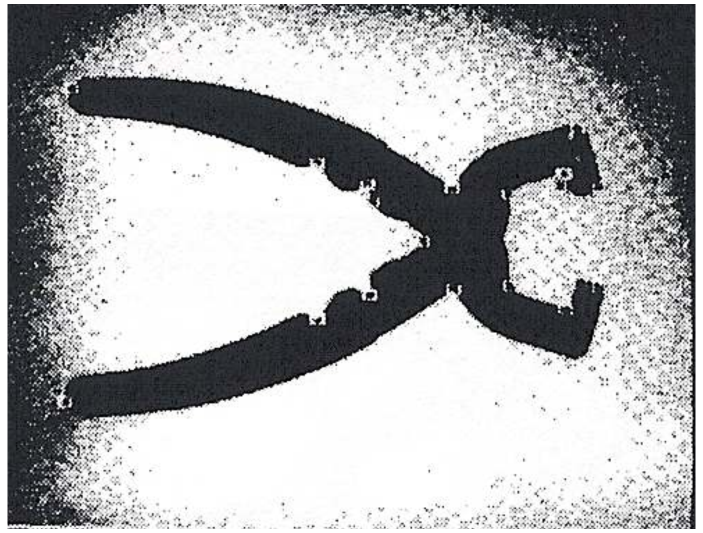
Example Video



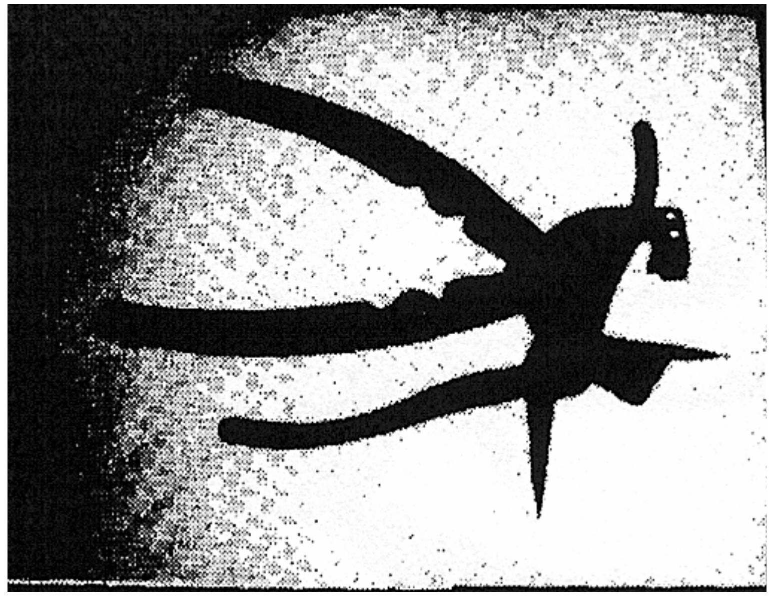
Application: Geometric Hashing

- Well-known technique for image matching
- Task:
  - Find (smaller) image (model) in large image (scene), including position/orientation/scaling
  - Preprocessing is OK
- Approach: consider only feature points
  - A.k.a. salient points, corner points, interest points

Find this image …



… in that image



140k pixels

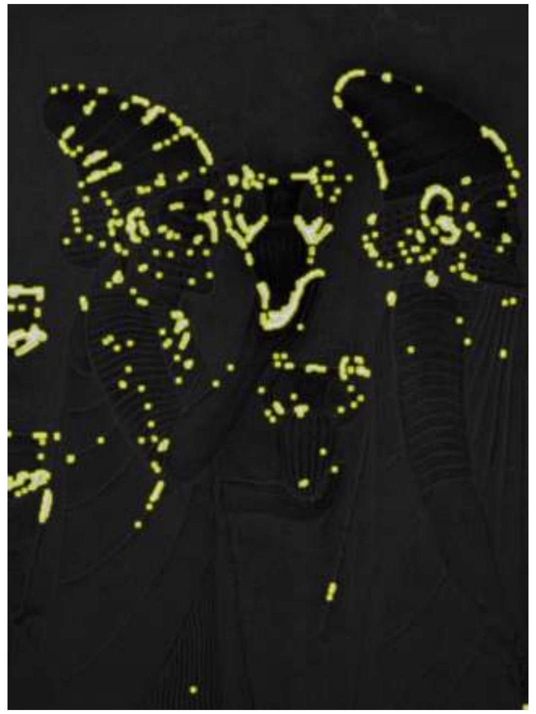
946 feature points (0.67%)

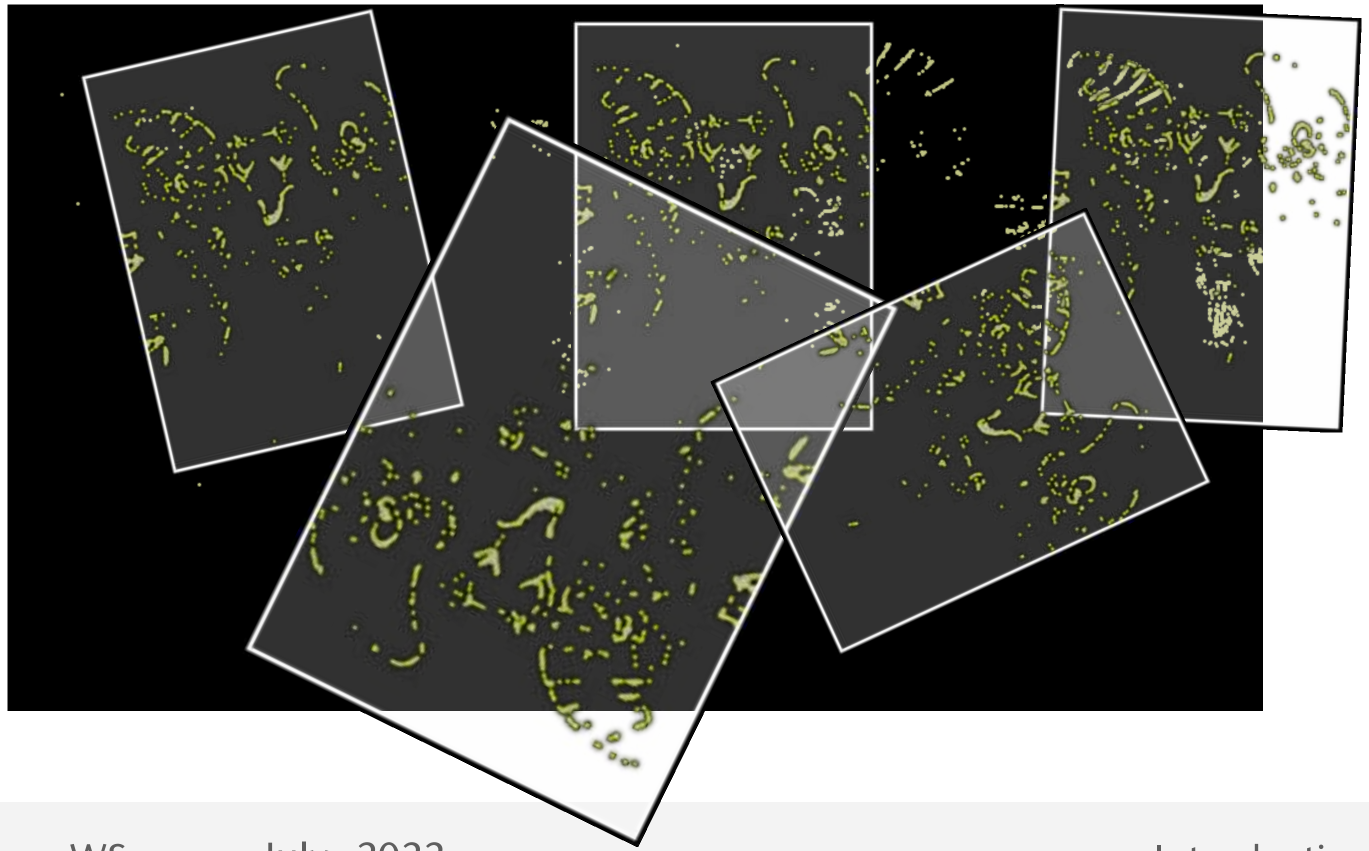
Bremen

Example

Model

Scene

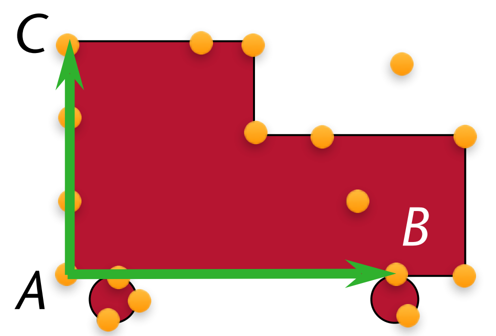


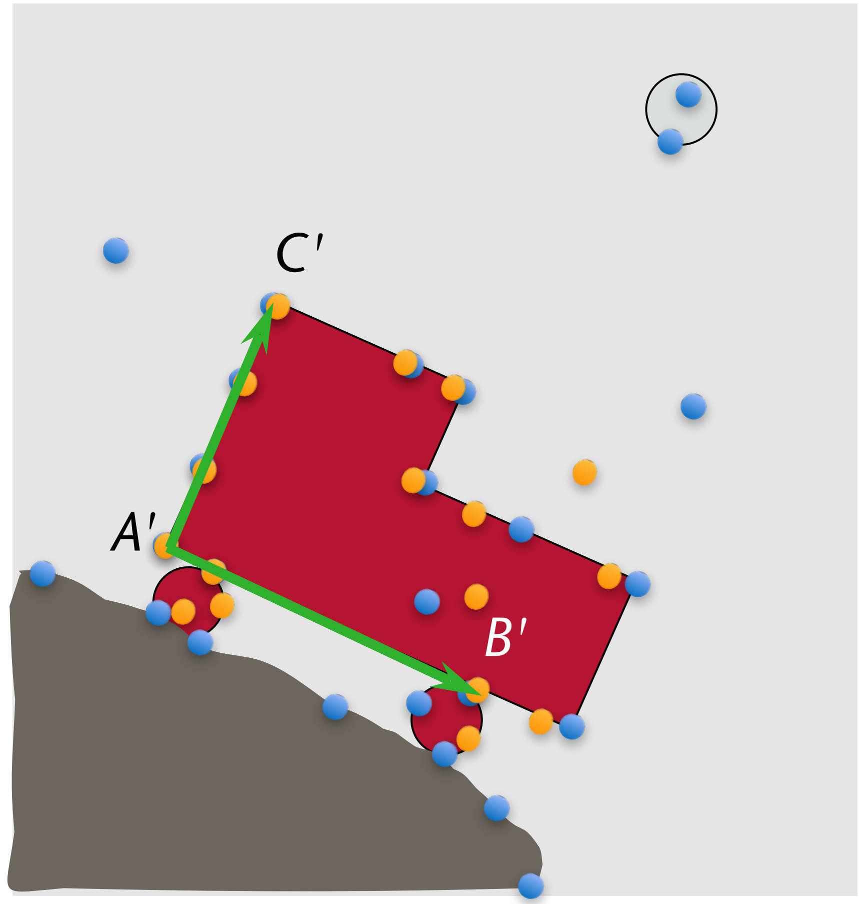


First (Naïve) Approach

- Preprocessing: build database of all models
  - One input image per model
  - Extract and store m feature points $\mathcal{F} = \{F_1, \ldots, F_m\}$ (per model)
- At runtime:
  - Extract n feature points in scene image $\mathcal{S} = \{S_1, \ldots, S_n\}$
  - Pick 3 non-collinear points $A, B, C \in \mathcal{F}$, and 3 points $A', B', C' \in \mathcal{S}$ (a 3x3 pairing)
  - Compute affine transformation mapping $A, B, C \longrightarrow A', B', C'$
  - Map all points in $\mathcal{F}$, calculate quality of match (e.g. RMSE)
  - Repeat with all possible 3x3 pairings
  - Choose optimal one (e.g., smallest RMSE)

Model





Is the model in the scene?
If so, where is it?

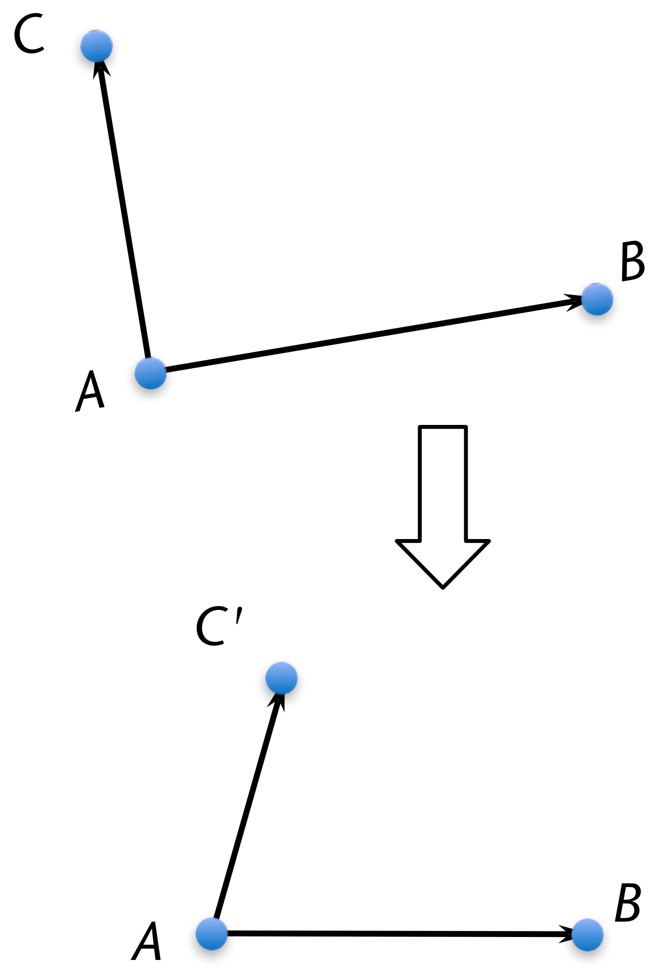
Digression: On Calculating the Affine Transformation

- Given A, B, C and A', B', C' - determine M s.t. $MA = A'$, $MB = B'$, $MC = C'$
- We are looking for a matrix M and vector T such that

$$\begin{pmatrix} a'_x \\ a'_y \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} a_x \\ a_y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

or, equivalently

$$\begin{pmatrix} a'_x \\ a'_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_x \\ a_y \\ 1 \end{pmatrix}$$



- The 3x3 pairing gives us

$$\begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ 1 & 1 & 1 \end{pmatrix} = \overbrace{\begin{pmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix}}^{M} \overbrace{\begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{pmatrix}}^{P}$$

- Multiplying by P^{-1} will yield M

Complexity of the Naïve Method

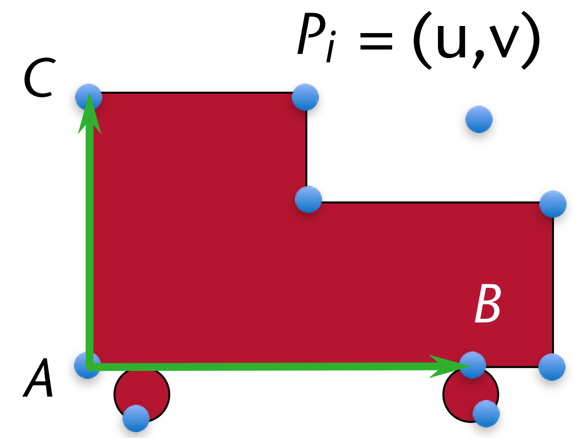
- There are $m^3 n^3$ possible 3x3 pairings
- Assume m $\approx 0.01n \longrightarrow m \in O(n)$
- Cost for computing one match (given aff. transformation) $\in O(m)$
  - In reality, it is worse, since for each model point, we need to find the closest scene point
- Overall complexity $\in O(n^7) \longrightarrow$ ouch!

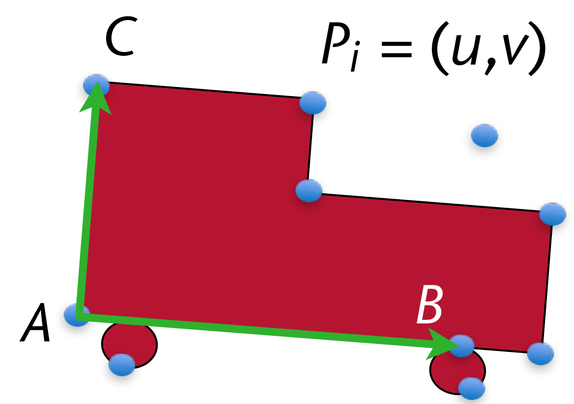
Geometric Hashing

- Idea: represent model in affinely invariant way
- Pick any 3 non-collinear points A, B, $C \in \mathcal{F}$; call this a basis
- All points $P_i \in \mathcal{F}$ can be represented wrt. this basis:

$$P_i = A + u(B - A) + v(C - A)$$

- Any affine transformation of the model will leave (u,v) invariant
    - Hence, (u,v)-representations are called invariants
- If only rotation & translation are allowed, then construct a basis as follows:
    - Pick any two points $A, B \in \mathcal{F}$ (not too close together)
    - Let $\mathbf{a}$:= normalize(B - A)
    - Let $\mathbf{b}$:= $(a_y, -a_x)$, i.e., the vector perpendicular to $\mathbf{a}$
    - Represent all other points as $P_i = A + u\mathbf{a} + v\mathbf{b}$





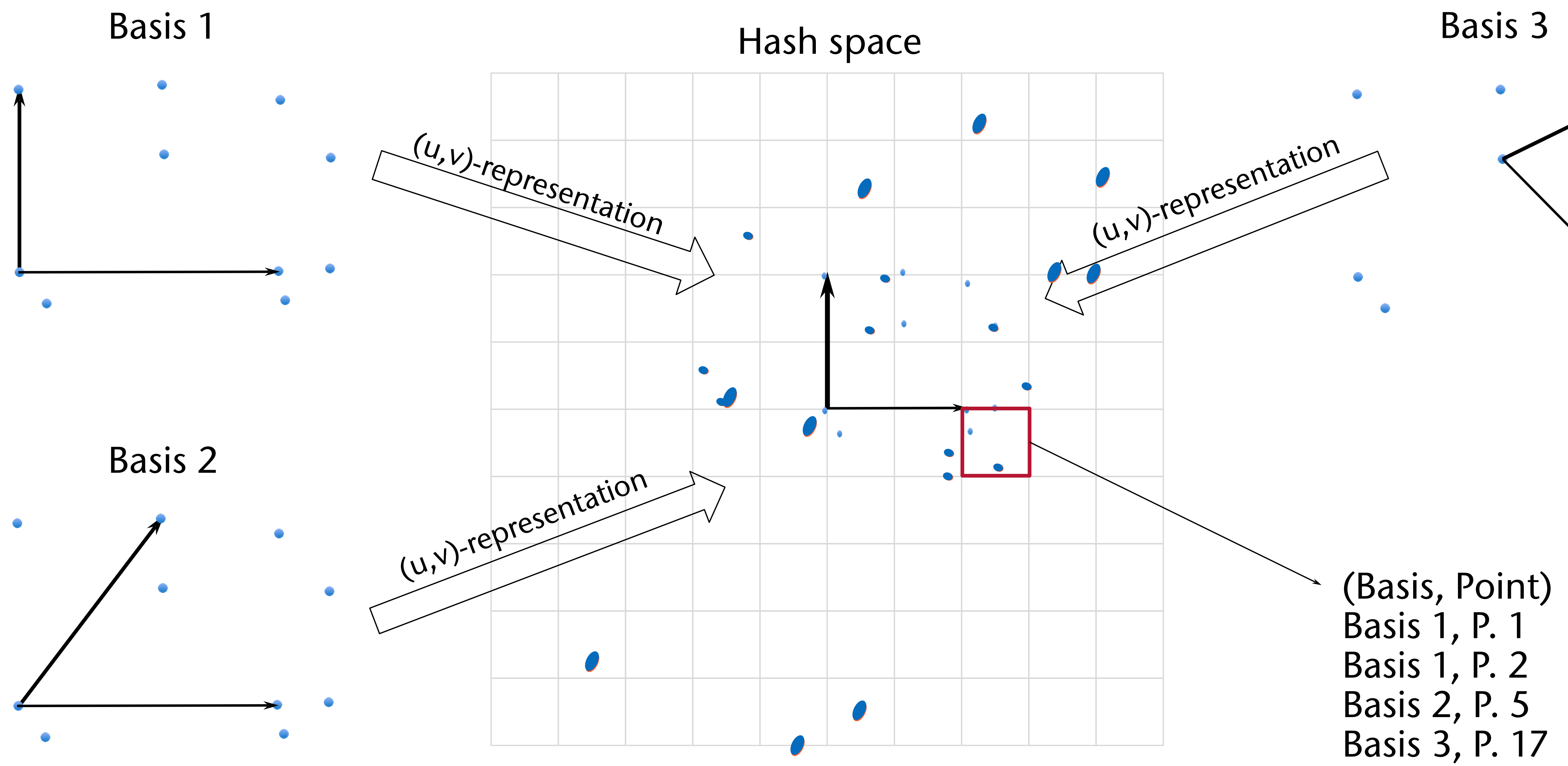
Preprocessing

- Fill hash table with (u,v)-representations of *all* feature points wrt. *all* possible bases:

```
forall bases E = (A, B, C) ⊂ ℱ :
   forall other points P ∈ ℱ :
      calculate (u,v) wrt. E
      convert u,v to integer coords (scale & round)
      store (P,E) with key (u,v) in spatial hash table
```

- Do this for all models M
  - Note: can even store *all* models this way in *one* common hash table
    $\longrightarrow$ store (M,P,E) with keys (u,v)
  - In the following: consider just one model (for sake of simplicity)
- Note: quantization of (u,v) provides actually some amount of robustness
  - Slight shifts of the feature points do not change their hash table slot (in many cases)

Example


Basis 1
Hash space
Basis 3
(u,v)-representation
(u,v)-representation
Basis 2
(u,v)-representation
(Basis, Point)
Basis 1, P. 1
Basis 1, P. 2
Basis 2, P. 5
Basis 3, P. 17

- Note: more models can be added dynamically to the hash table

- Complexity of preprocessing: O(m^4) per model

Recognition

- First phase: detect all feature points in the scene image $\longrightarrow S$
- Second phase: hypothesis generation = maintain number of "votes" for each basis in the *model*
  - Result: a histogram over all possible bases, one bin per basis of the model, counting the number of votes for each basis
- The algorithm:

```
forall bases E ∈ S :
   clear histogram of votes
   forall other points P ∈ S :
      calculate (u,v) wrt. E
      convert u,v to integer coords (scale & round)
      forall entries (B,X) in slot (u,v) of hash table:
         increment vote count of histogram bin of basis B
   forall bases B where #votes > threshold:
      record hypothesis (B,E)
```

- Reasoning behind the algorithm:
  - If E happens to be the basis where the model is present in the scene $\longrightarrow$ there is a "matching" basis B in the model
  - Let M be the affine transformation from B to E
  - For many points in $\mathcal{F}' = M(\mathcal{F})$, there will be a nearby point in $\mathcal{S}$
  - Therefore, many points of the scene image will fall into hash table slots containing at least one entry $(B,*)$
  - Therefore, B will garner more "votes" than other bases of the model
- Note:
  - Every hypothesis (B,E) provides an affine transformation M from model space into scene space, such that "many" points in $M(\mathcal{F})$ are "close" to a point in $\mathcal{S}$
  - Meaning of "many" = "> threshold"
  - Meaning of "close" = "< diameter of grid cell"

- Third phase: test the hypotheses
- Algorithm:

```
forall hypotheses (B,E):
   compute affine transformation M from B to E     // (*)
   transform all model points → ℱ' = M(ℱ)
   let score of (B,E) = RMSE(ℱ',𝒮)
choose the hypothesis (B,E) with the highest score
```

- Note: in the RMSE, we consider the closest point in $\mathcal{S}$ to each point in $\mathcal{F}$
  - Use spatial the hash table over $\mathcal{S}$ for that, or a kd-tree (see comp. geometry)
- Note on step (*):
  - We could just use the method from slide 13 (aff. trf. for 3x3 pairing)
  - More robust is a least squares method (omitted here)
    - From hypothesis generation, we already have a $k \times k$ pairing

- Complexity of recognition $\in O(n^4)$

- In a way, the hash table serves as an acceleration data structure for finding nearest neighbors quickly
- Ideas:
  - Use kd-trees, or
  - Consider neighbor cells in the hash table, too

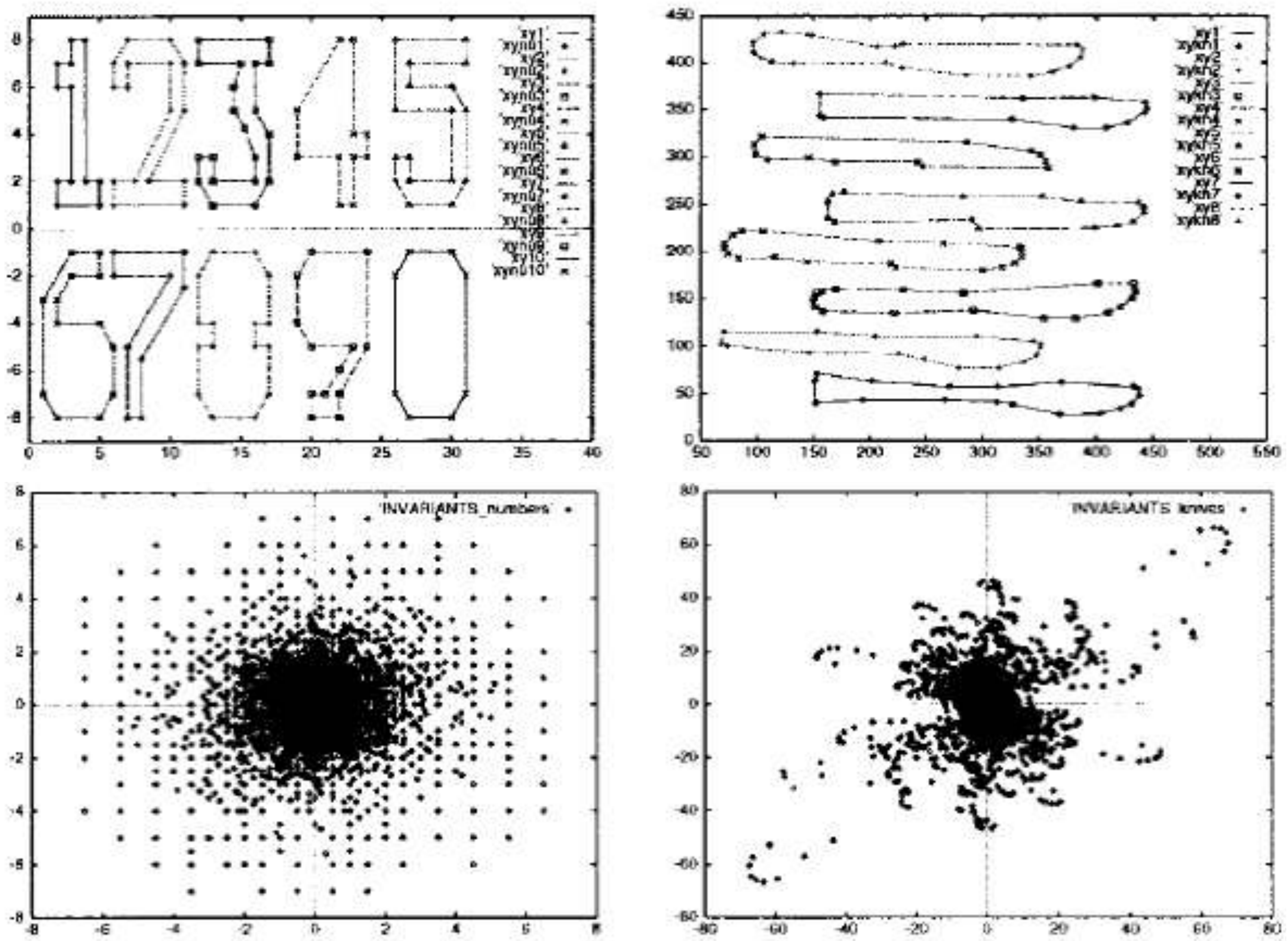
Improvement in Case of Non-Uniform Distribution of Feature Points

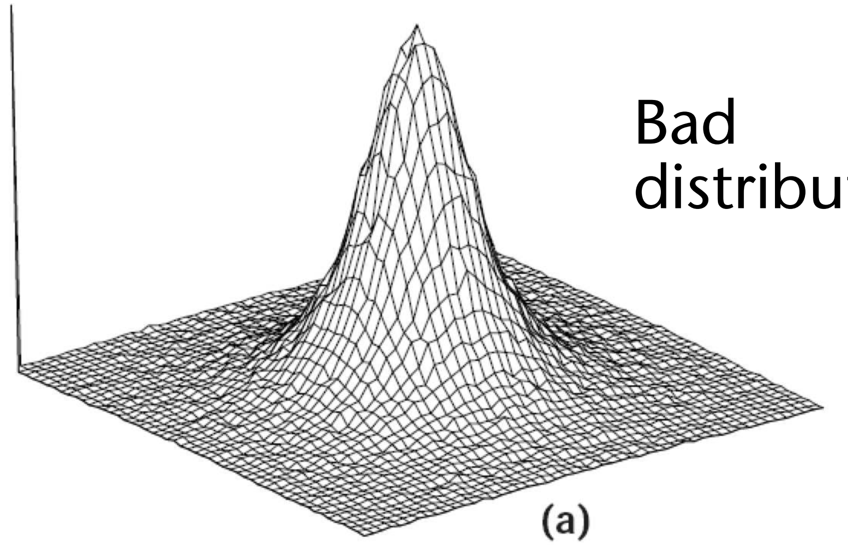
- The distribution of the feature points in (u,v) space might be highly non-uniform $\longrightarrow$ lookup in hash table is no longer O(1)!

Letters Knives

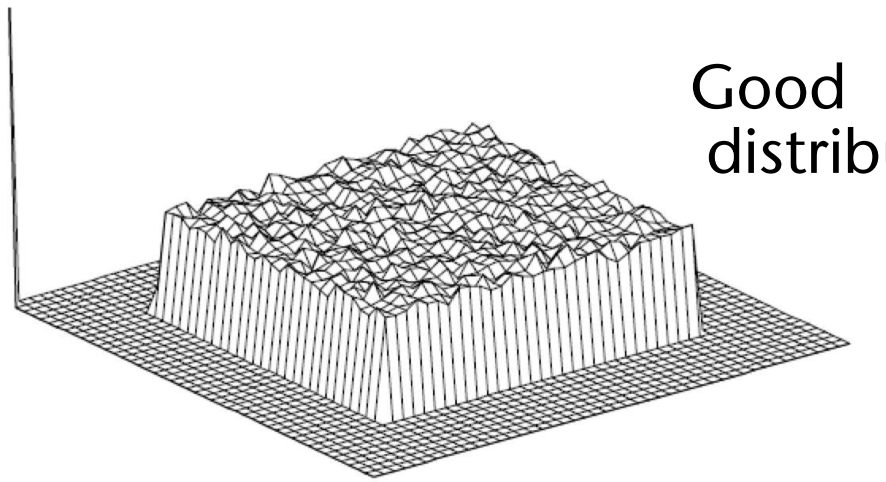
Cartesian space

Hash table



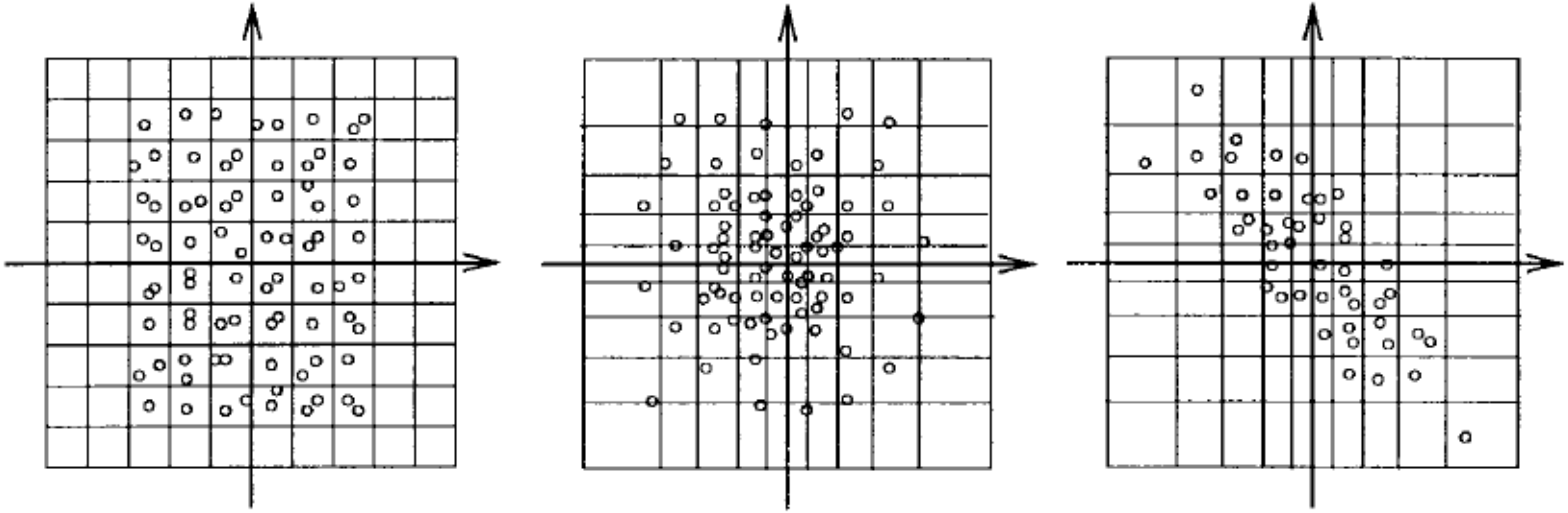


Bad distribution



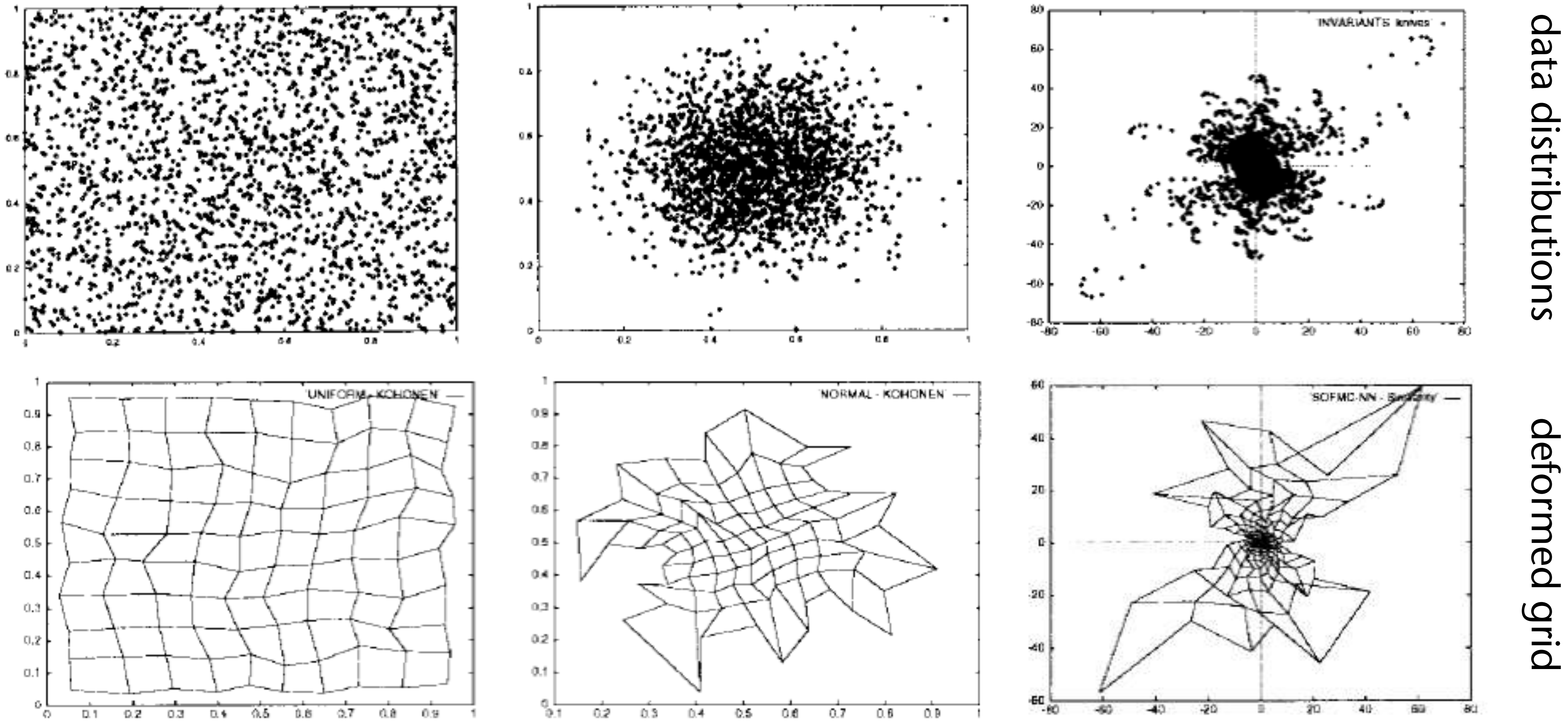
Good distribution

- One approach: make the size of the voxels proportional to the density of the data

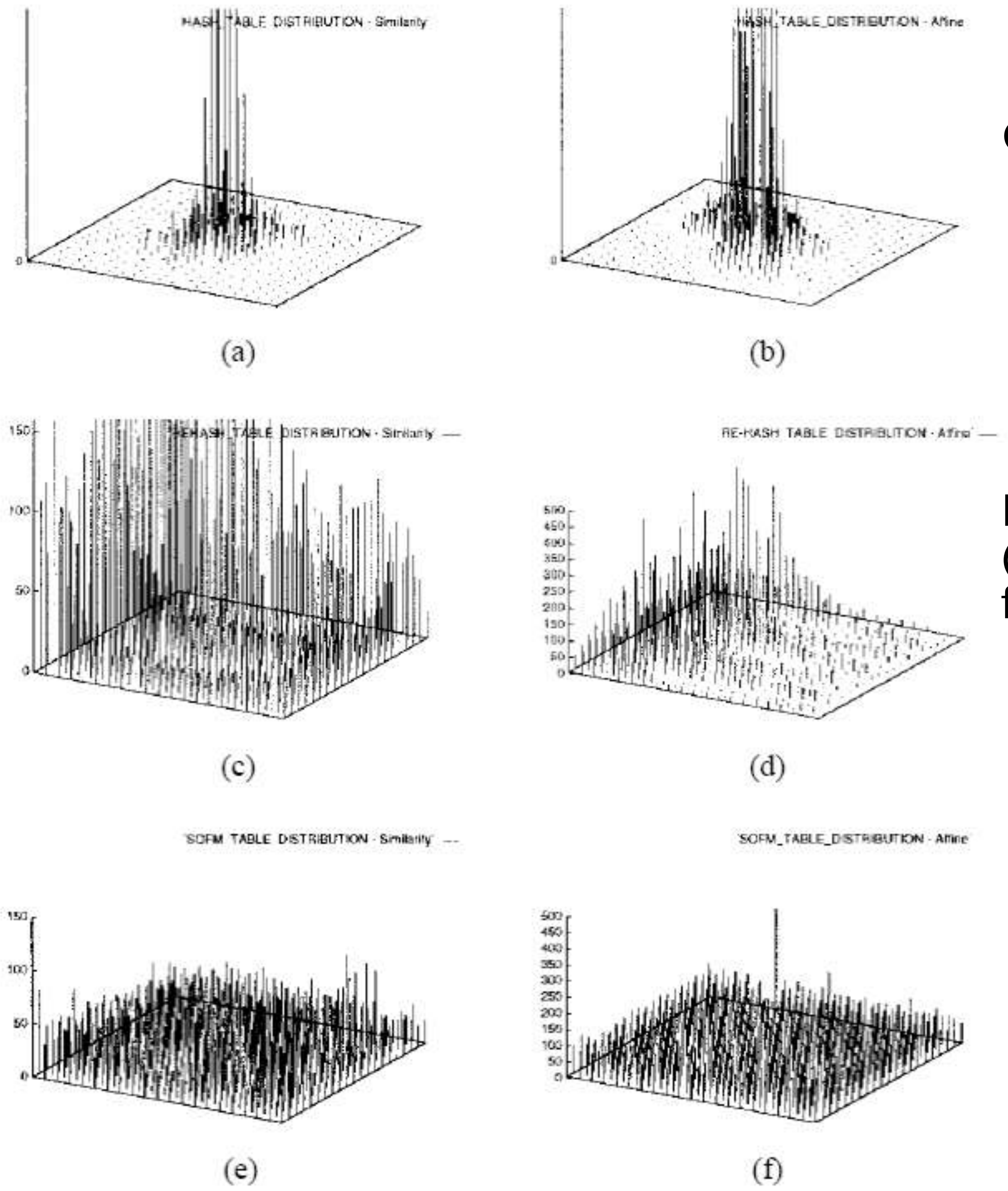


Other Approach: "Learn" a Good Spatial Partitioning

- Consider the background grid as "elastic" net that deforms based on the density of the data
- Kohonen neural networks do just that



Results



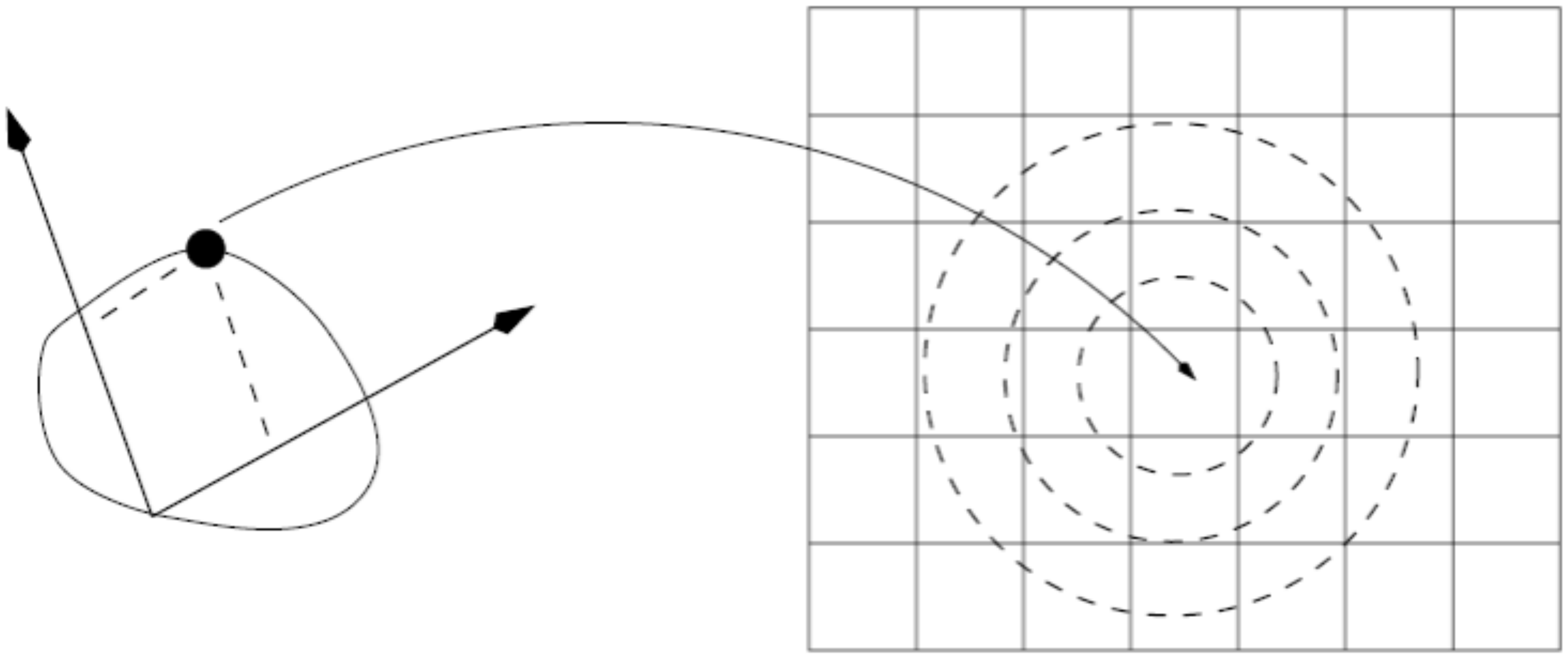
Original

Rehashing
(using a transfer function)

Learning

Noise

- Experience shows: performance of Geometric hashing degrades rapidly for cluttered scenes or in the presence of moderate sensor noise (3-5 pixels)
- Possible solutions:
  - Make additional entries during preprocessing (increases storage)
  - Cast additional votes during recognition (increases time)



Another Solution for Noise

- Observations:
  1. The larger the separation of basis points, the smaller the effect of noise offsets on the final slots in the hash table
  2. The closer a point is to the origin of the basis, the smaller the effect of noise offsets on the final slot in the hash table
  3. Areas in uv-space with high density of feature points contain less information than areas with low density $\longrightarrow$ hash table cells with many entries contain less information than cells with few entries
- Weight the vote of hash table entries based on these criteria

Massively Parallel Geometric Hashing

- Input: color image
- Feature point detection (both images):
  - One thread per pixel
  - Apply e.g. Sobel operator at each pixel (or, ORB, BRIEF, etc.)
  - If above threshold, then output Cartesian coords
  - Compact output array $\longrightarrow$ m feature points
- Preprocessing (fill hash table):
  - One thread per basis $\longrightarrow$ m^3 threads
  - Each thread iterates through all other feature points: calculate (u,v), store in hash table
  - Optionally: just consider random subset of bases

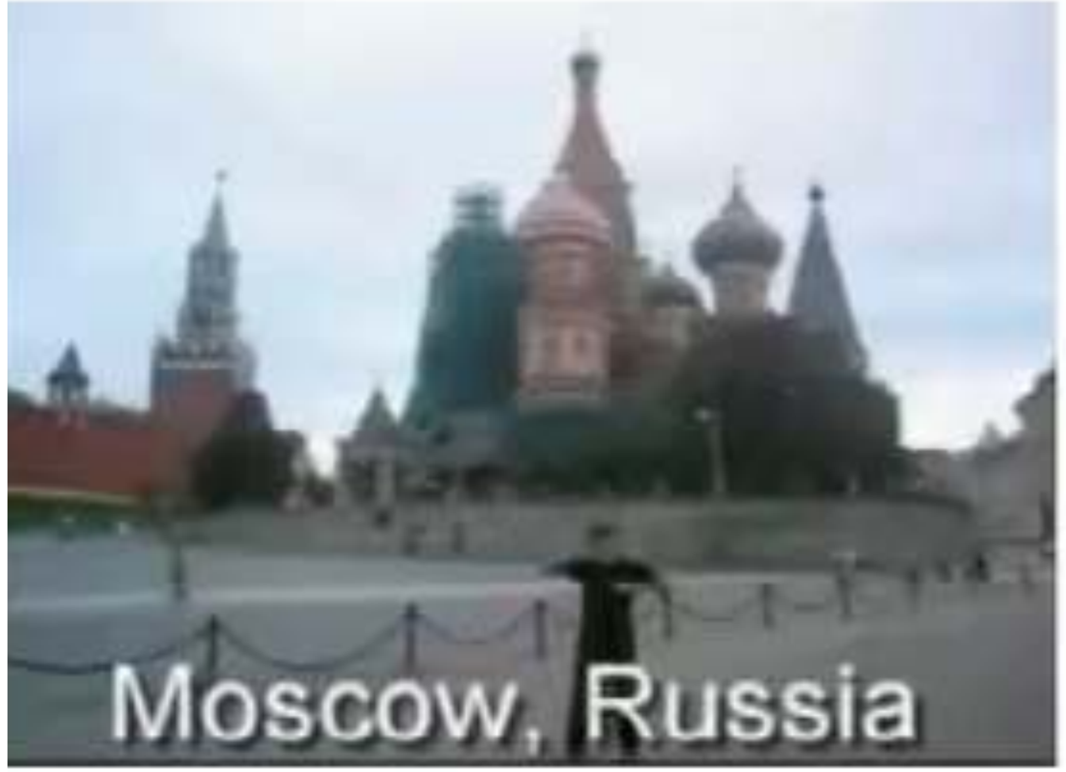
Object Recognition

- One thread per basis E in query image (n^3 threads, or random subset), each one iterates over all *other* feature points
- For each other feature point (u,v): iterate over all values B stored in the hash table slot for key (u,v)
- For each such basis B: cast a vote for correspondence (B,E)
- Store votes in a matrix V of size $m^3 \times n^3$
  - (Or less in case of random subsets of $\mathcal{F}^3$ and $\mathcal{S}^3$, resp., for the bases)
- Compact V: output all basis pairs with #votes > threshold
  - One thread per element, or one thread per row

Example

Model

Scene

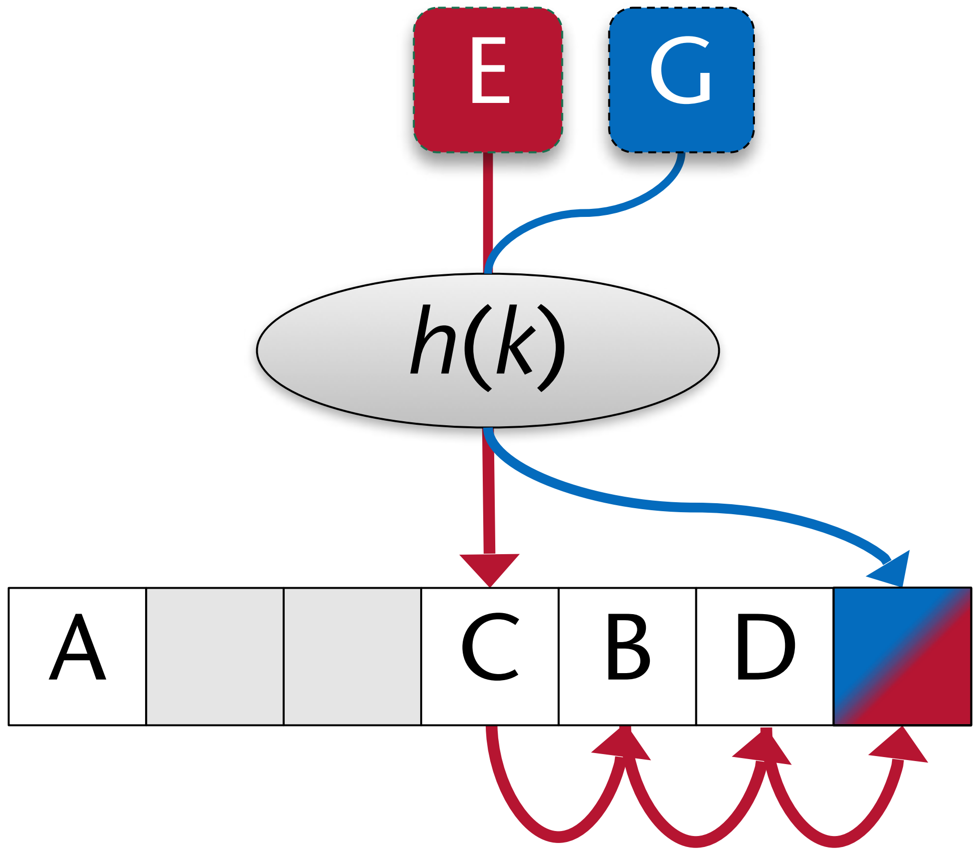


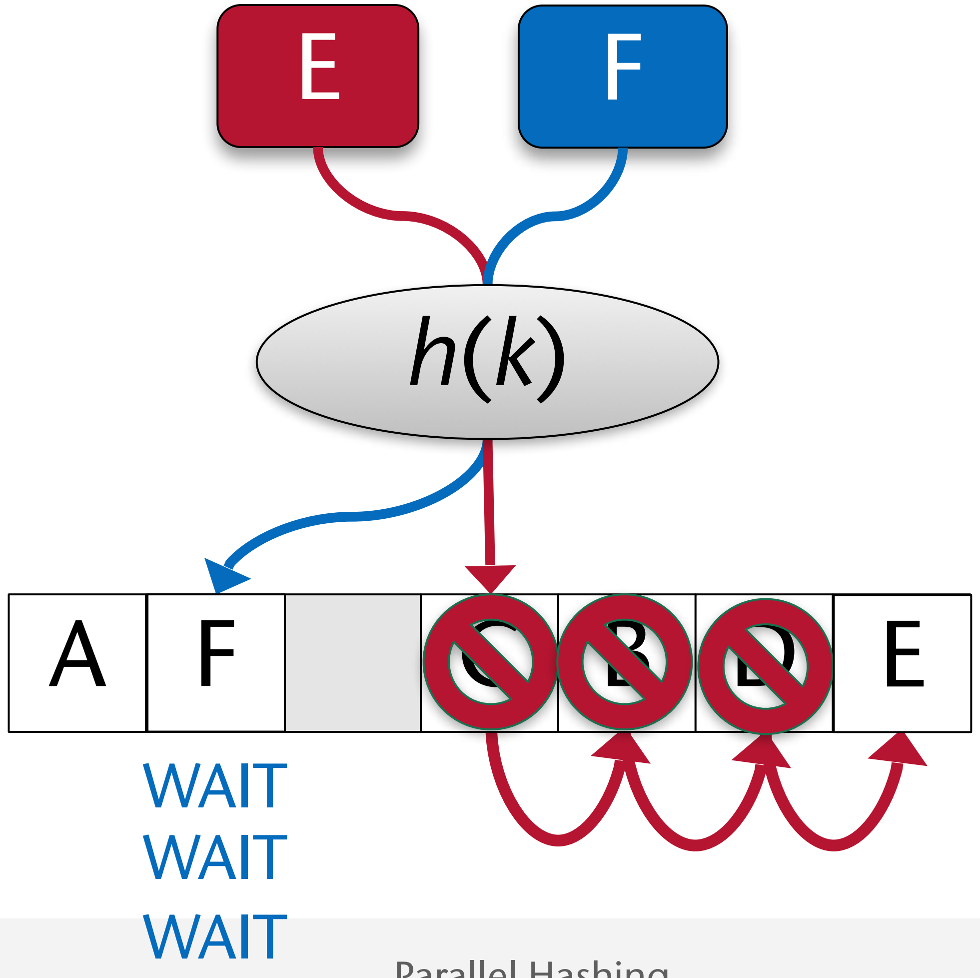
[Alcantara, 2009]

Traditional Hashing

- Probing for resolving collisions in hash table
  - E.g., linear or quadratic probing, or double hashing
- Parallel insertion requires serialization (locking of the hash table)
- Consequence: all threads in a block wait until the lock-holding thread has finished

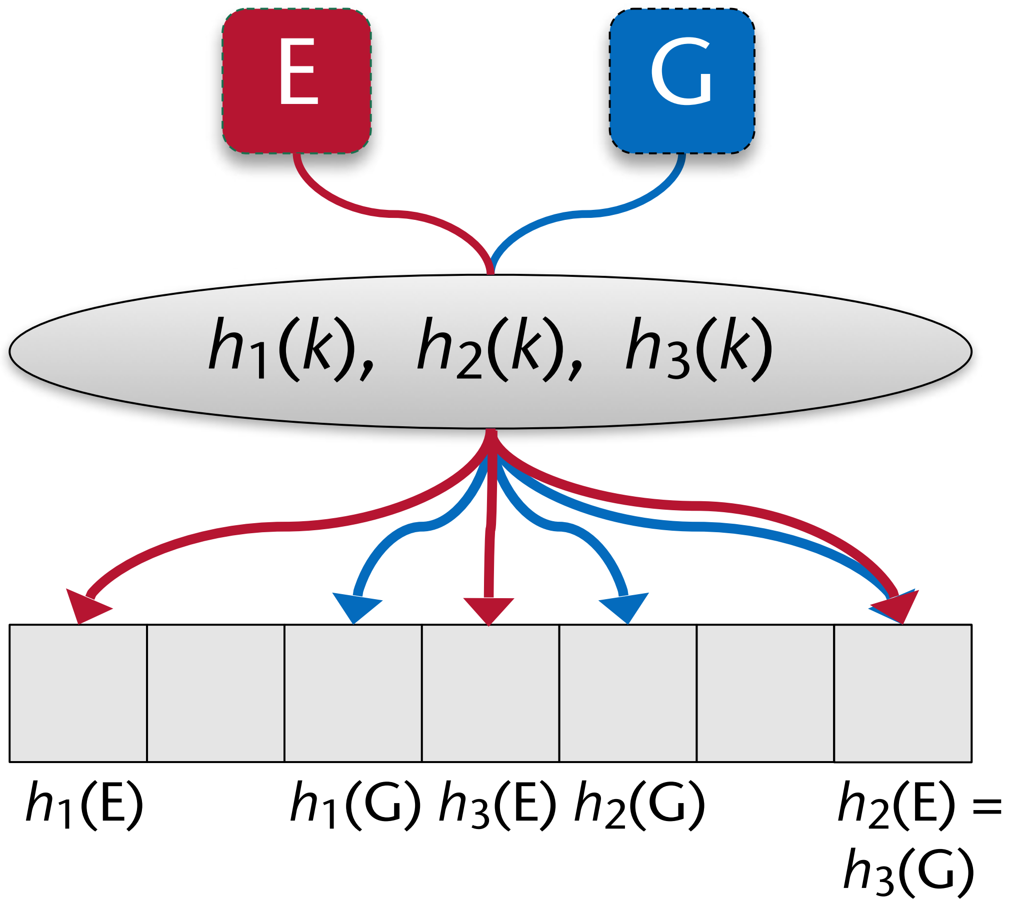
➢ Long probing sequences are bad for the overall performance of *all* threads in the block



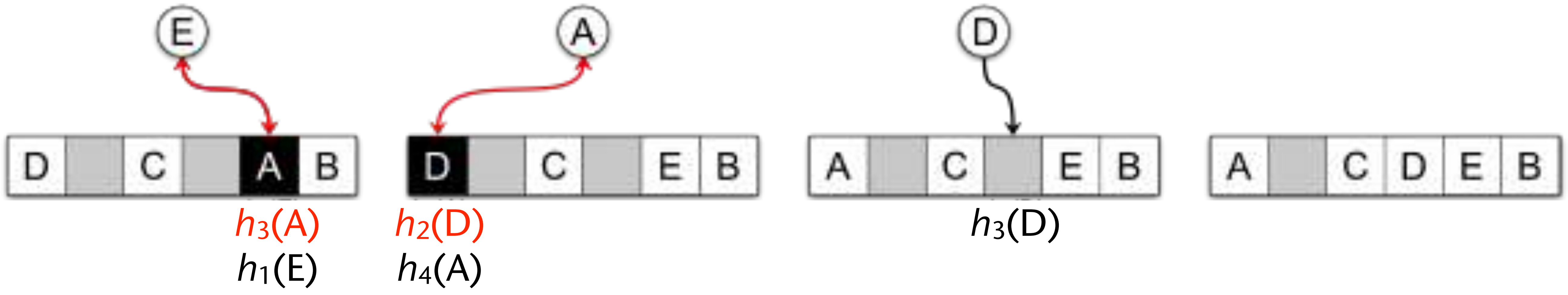


Cuckoo Hashing

- Fact: parallel hash table accesses are almost always *uncoalsced*
  - Consequence: minimize number of memory accesses
- Idea:
  - Each key k gets mapped to a number of different hash table slots
  - Instead of probing: use a number of hash functions $h_1, \ldots, h_f$



- Example:



  - Note how keys can get evicted (hence the name) $\longrightarrow$ eviction chain
  - Hash functions are used in round-robin fashion
- In practice, "simple" hash functions work well:
  - Randomly generate $h_i(k) = a_i k + b_i \mod p \mod m$

    with p = 334 214 459, m = number of slots, and randomly generated constant a_i, $b_i \in [0,p)$
  - Variant: XOR instead of multiplication, p = 4 294 967 291 (= 2^{32}-5)

- Advantage: even in the worst case, lookup time is O(1) !
- Parallelization: one thread per key during insert/lookup
- Note:
  - Threads in a block still need to wait for all others to finish
  - Threads do not need to lock hash table (except for the atomic swap)
- Problem: insertion could fail
- Solution: stash
  - During insert, a thread follows a "chain of evictions"
  - If this gets too long (or ends in a cycle), give up $\longrightarrow$ store key in stash
  - Stash = simple array, or hash table with very low load factor
  - In practice, only 5 keys hit the stash

The Algorithm

- Store key and value contiguously in memory
  - Memory access is better coalesced
  - Allows to use single atomic swap operation for both

```
class HashEntry
{
   uint32 key;
   uint32 value;
   ...
}
```

- Initialization of hash table: fill all slots with entries (0xFFFFFFFF, 0)

Insertion Into the Hash Table

```
fct insertIntoHash( key, value ):                 // can be called in parallel
entry = HashEntry( key, value )
slot = hash_fct( 0, key )                         // = h0(key)
repeat max_tries:
   entry = atomicExch( & table[slot], entry )
   key = entry.key
   if key == EMPTY:                               // found an empty slot
      return true
   // else, entry got evicted
   for j = 0 .. n_hash_fct-1:                     // = f from previous slide
      if hash_fct(j, key) == slot:
         break                                    // exactly one j must break
   j = (j+1) mod n_hash_fct                       // try "next" hash fct
   slot = hash_fct(j, key)

try to append entry to stash (or insert if stash is a hash table)
if that fails:
   signal failure to caller,
   rebuild whole hash table with other random hash functions
```

Retrieval

```
fct retrieveFromHash( key ):             // can be called in parallel
loop j = 0 .. n_hash_fct-1:
   slot = hash_fct( j, key )             // = hj(key)
   if table[slot] == key:
      return table[slot].value
   if table[slot] == EMPTY:              // short-circuit
      break
// key is not in main hash table
if stash was not used during construction:
   return NOT_FOUND
check stash
```

A Quick Excursion into Theoretical Computer Science

- Question: what is the probability that cuckoo hashing works?
- Rephrasing:
  - Let keys = $K = \{x_1, \ldots, x_n\}$, slots = $S = \{1, \ldots, m\}$, $m > n$
  - Assume $m = c \cdot n$, $c > 1$ fixed (e.g., $c = 1.4$)
    - $1/c$ = load factor (I'll call c a load factor, too)
  - For each x_i, there is a given (random) set of permissible slots:

$$S_i = \{j_1^i, \ldots, j_f^i\} \subset S, \text{ where } j_l^i = h_l(x_i)$$

  - Can we find a mapping $\tau : K \to S$ such that all $\tau(x_i)$ are mutually different, and $\forall i : \tau(x_i) \in S_i$?
  - What is the probability of finding such a τ ?

- Trick 1: associate a rectangular matrix M with the keys and slots
  - Every row corresponds to one key, every column corresponds to one slot in the hash table
  - For each key x_i, we fill its row in M as follows:
    write a "1" in columns $j_1^i, \ldots, j_f^i$, and 0 everywhere else
  - So, M is a $n \times m$ matrix over {0,1} (more columns than rows)

- Example:
  - $n = 4$ keys, $m = 7$ slots, $k = 3$ different hash functions
  - $S_1 = \{2, 4, 5\}$
  - $S_2 = \{1, 2, 6\}$
  - $S_3 = \{3, 4, 7\}$
  - $S_4 = \{1, 3, 6\}$
  - Matrix

$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

- Trick 2: associate a linear system of equations with the S_i
  - The system is

$$M\,z = b$$

    where all variables are only 0's and 1's , and addition is modulo 2, i.e., arithmetic is over the field $\mathbb{Z}_2$ (so we have, for instance, an inverse)
  - Choose $b \in \{0,1\}^n$ randomly
    - Exactly which b is not important, important is its *randomness*
  - In the end, we won't care about the solution z (if any)

The chain of arguments

- If the system has a solution (1)
  $\Longrightarrow$ M has maximal rank in rows (i.e., all rows are linearly independent) (2)
  $\Longrightarrow$ M has also maximal rank in columns = n
  $\Longrightarrow$ we can pick n columns from M and form square matrix M' with $\det(M') \neq 0$
- Consider the Leibniz formula for the determinant:

$$\det(M') = \sum_{\sigma \in \mathrm{Perm}(n)} \mathrm{sign}(\sigma) m'_{1,\sigma(1)} m'_{2,\sigma(2)} \cdots m'_{n,\sigma(n)}$$

- Remember the special contents of M, and remember we calculate in $\mathbb{Z}_2$!
- So, $\det(M') \neq 0 \implies$ at least one of the product terms must equal 1
- Take the σ that produces that term (or one of them)

- "Translate" the permutation σ into a mapping τ:

  every σ(i) corresponds to a column in M', which was an original column in M

  $\longrightarrow$ assign that column number to τ(i)
- So, the term $m_{1,\tau(1)} m_{2,\tau(2)} \cdots m_{n,\tau(n)} = 1$
- In other words, every $m_{i,\tau(i)} = 1$
- Remember that the rows represent the sets of possible slots for the keys
- So, we have found one slot per key out of the permissible ones and they don't collide $\longrightarrow$ cuckoo hashing works
  - For this set of keys, and this set of hash functions!

Example continued

- We can find 4 linearly independent columns (over $\mathbb{Z}_2$)

$$M = \begin{array}{c} \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \\ \begin{array}{ccccccc} \uparrow & \uparrow & \uparrow & & \uparrow & & \\ 1 & 2 & 3 & & 5 & & \end{array} \end{array} \quad \Longrightarrow \quad M' = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

- The product in the determinant formula with $\sigma(1)=4, \sigma(2)=2, \sigma(3)=3, \sigma(4)=1$ is non-zero
- This translates to $\tau(1)=5, \tau(2)=2, \tau(3)=3$ und $\tau(4)=1$ for M
- Indeed, 5 is in S_1 (possible slots for key 1), 2 is in S_2, 3 in S_3, 1 in S_4 $\longrightarrow$
- We can store all keys in the hash table in one of their permissible slots

- Let M be a randomly chosen $n \times m$ matrix, but with the additional constraint that there are exactly f 1's in each row. Let b be a randomly chosen $\{0,1\}$ vector of length n.
  What is the probability that the system
  $$Mz = b$$
  has a solution?

- Theorem (w/o proof):
  If $m = c \cdot n$, and $c > c_f$, then such a system has a solution with high probability.

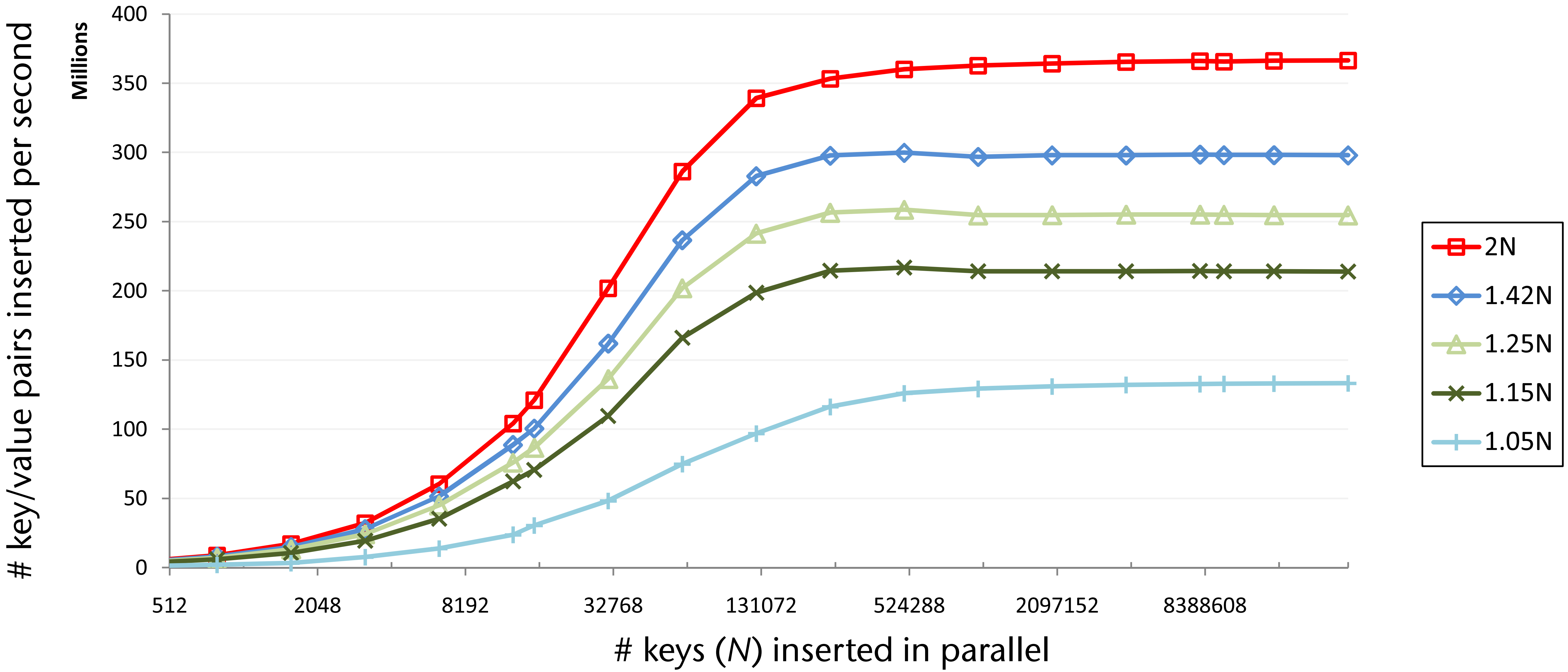
- The meaning of "high probability":
  as n (and, thus, m) go to infinity, the probability approaches 1

- Theoretical and practical bounds for the load factors, *c*, i.e., #slots ≥ *c*×#keys:

# hash fct *f*	c_{theor}	$c_{\text{practical}}$
2	-	2.1
3	1.089	1.1
4	1.024	1.03
5	1.008	1.02

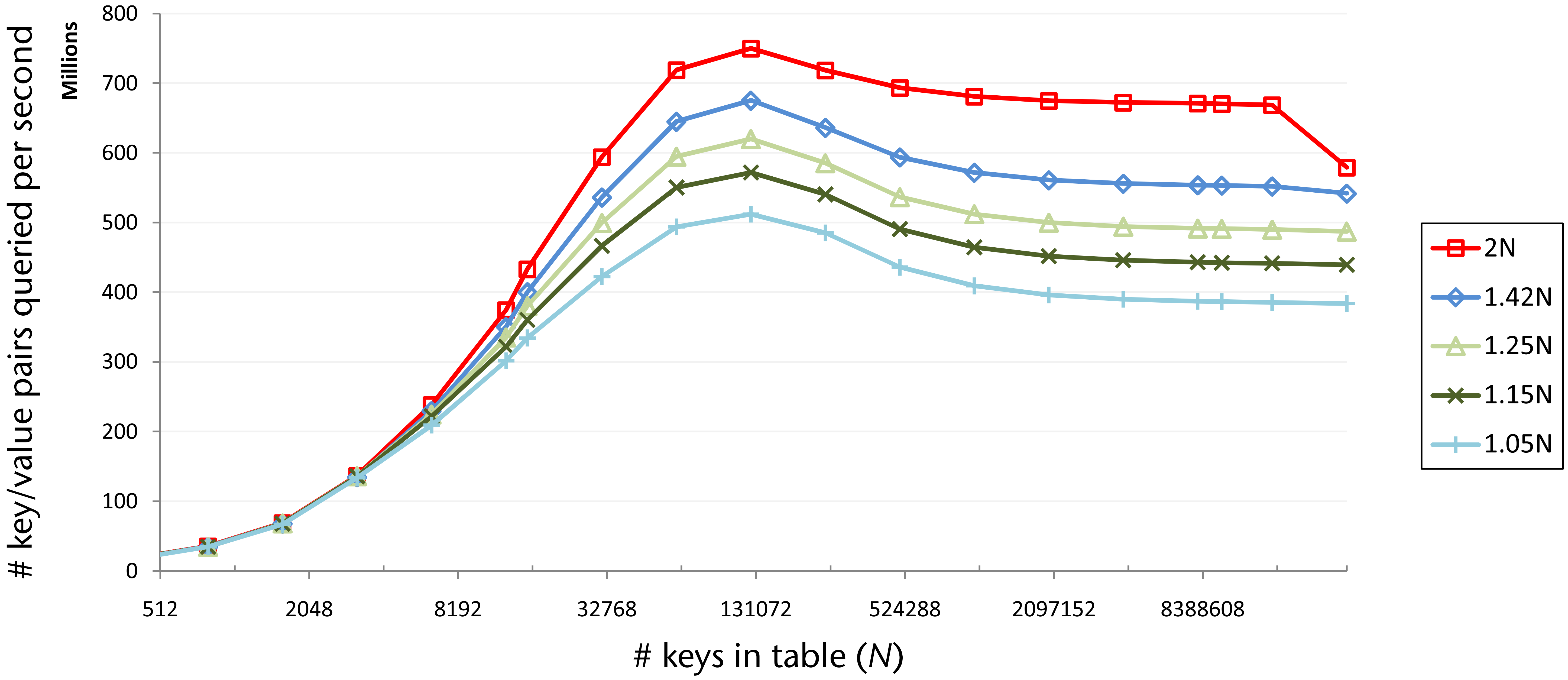
Performance of Cuckoo Hashing

- Performance for *insertion* depending on hash table load factor and number of keys (on GTX 470, using 4 hash functions):

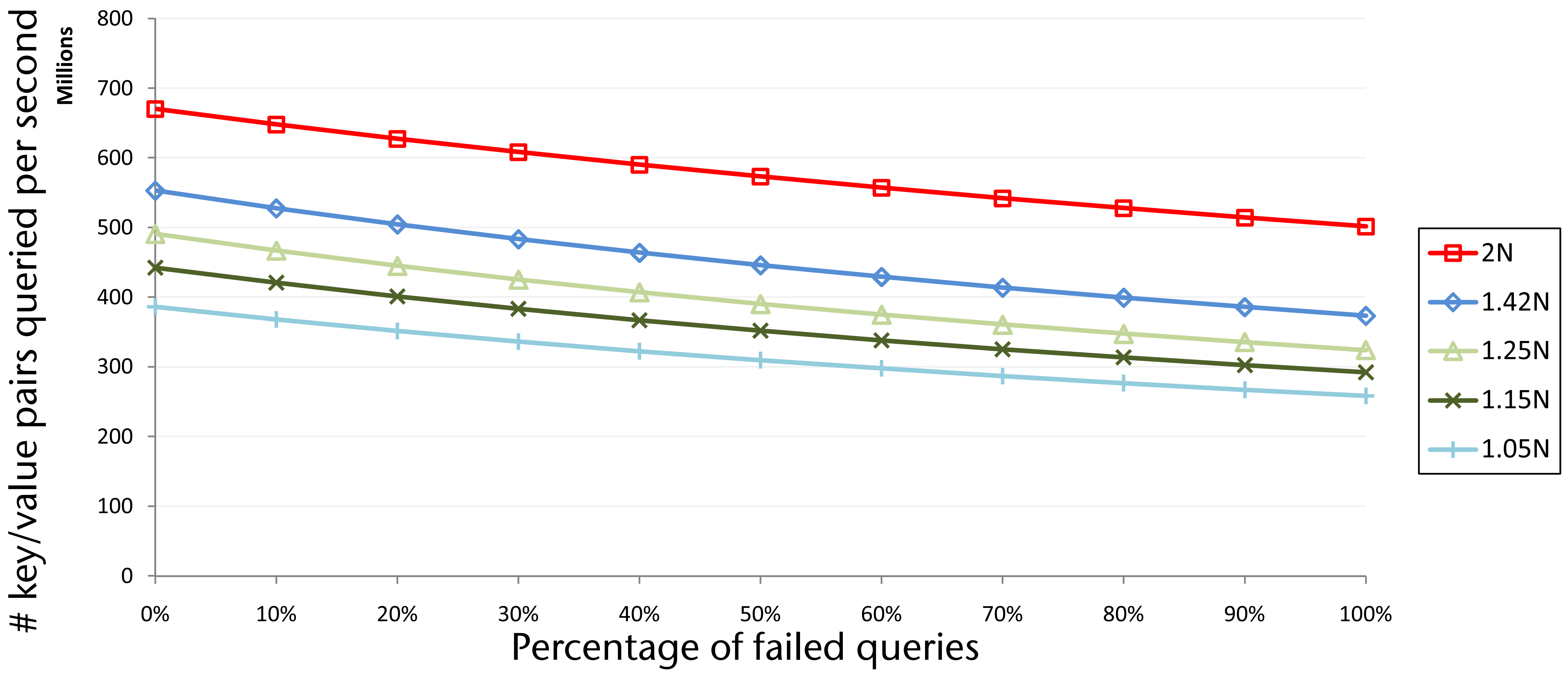


[Alcantara 2011]

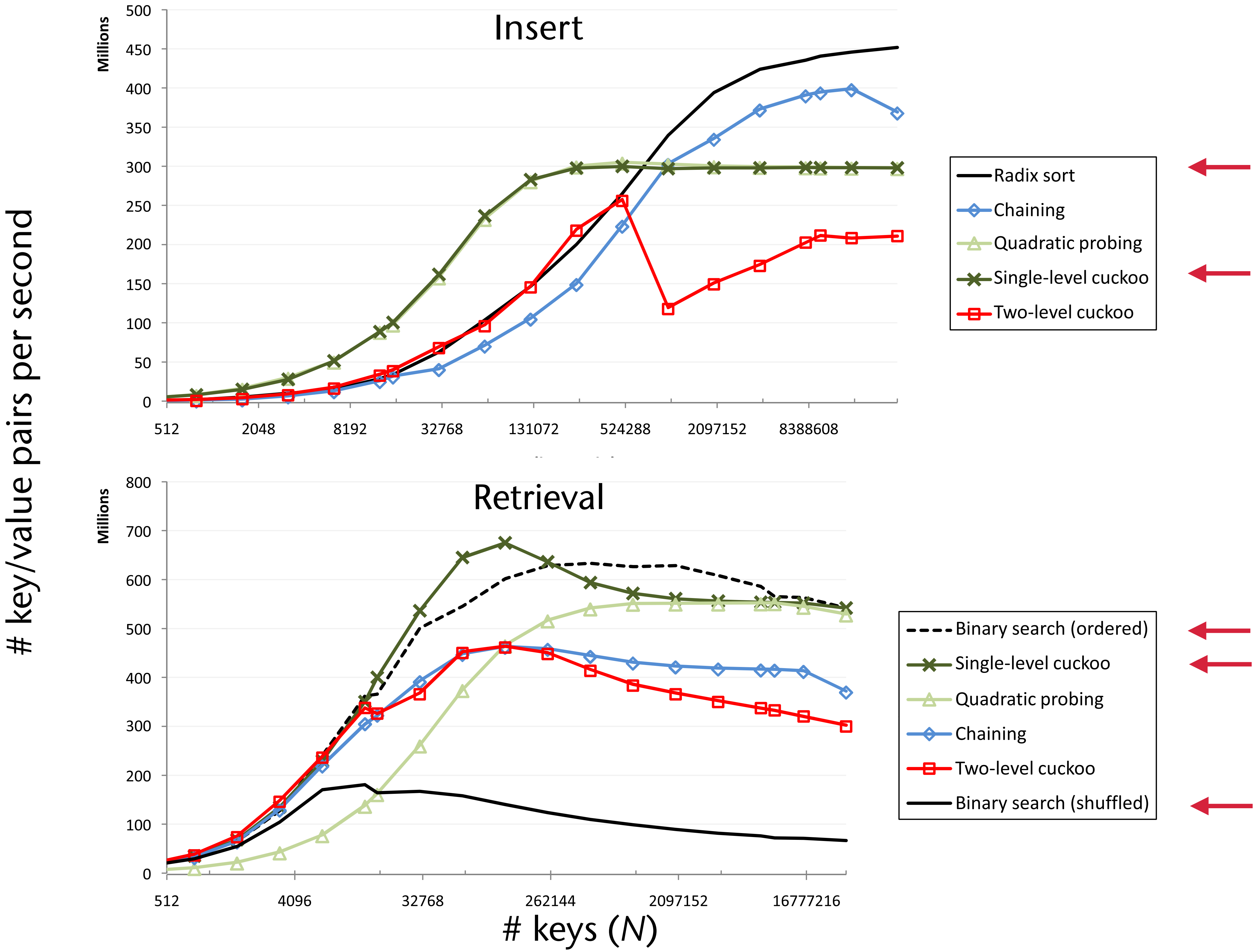
- Performance for *retrieval* depending on hash table load factor and number of keys (on GTX 470, using 4 hash functions, no failed keys):



- Performance depending on percentage of *failed* queries (key is not in hash table), N = 8.4M keys, GTX 470, 4 hash functions
  - Failed query = 4 regular probes into hash table, plus 1 probe into stash (implemented as hash table)



Comparison with a sorted array (#slots = 1.42×#keys)



Remember: *N* keys are inserted/queried in parallel by *N* threads

Ideas for Further Investigation

- Store the hash function ID with the key in the slot (e.g. in a few bits)
  - If it gets evicted, the thread doesn't have to re-compute this ID
- Is it possible to utilize shared memory for the build phase?
  - Warning: Alcantara tried it
- Is it possible to optimize the hash functions?
  - Choose a set of random hash functions, test insertion with a large number of random keys, determine length of eviction chains
  - Try a number of other hash function sets, pick the "best" one
- Instead of using the hash functions in round-robin fashion, randomize this part, too
  - Theoretical question: how does that change probability of success?
- More hash functions hurt, but only because of global memory access $\longrightarrow$ can we use 2 bytes next to a slot for h_{i+1} ?