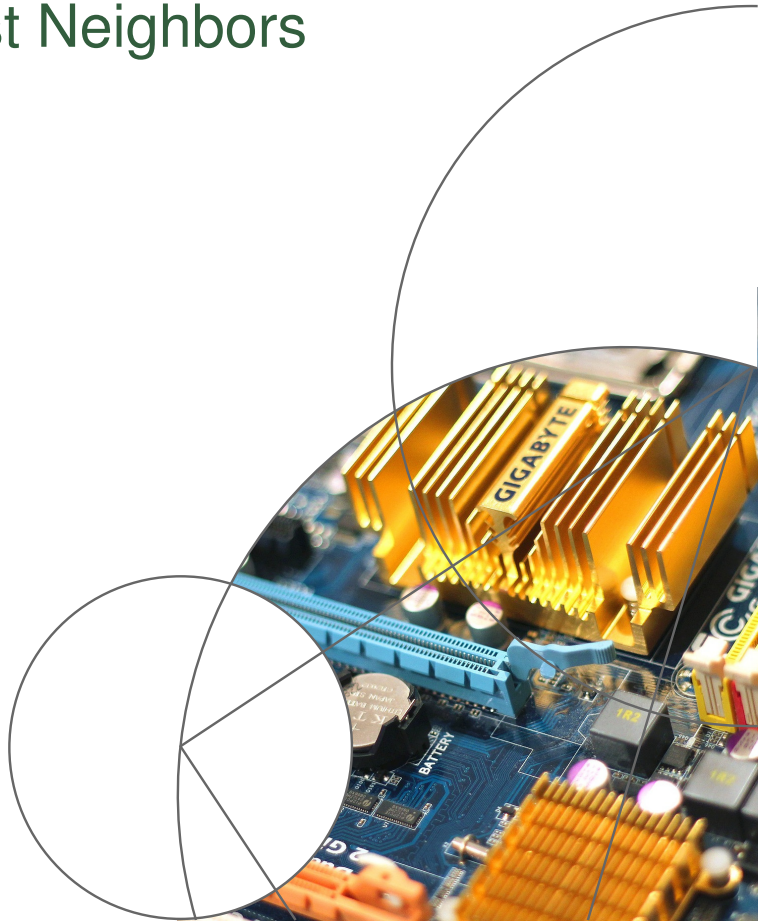
L2 – Computing & Large-Scale Nearest Neighbors

Large-Scale Data Analysis

Fabian Gieseke

Image Group
Department of Computer Science
University of Copenhagen

Universitetsparken 1, Room 1-1-N110
fabian.gieseke@di.ku.dk



25 April 2019

Comments

1. If you don't have a team yet: Let's meet during the break!
2. Today: Practical Sessions, 12:15-14:00
   - Tutorial VirtualBox (60 minutes)
   - Help Google Colab (getting the data to the instances, Google Drive)
3. Next Lectures
   - Computing & Large-Scale Nearest Neighbors
   - Introduction to TensorFlow
   - Large-Scale Least-Squares & Large-Scale Random Forests
   - Boosted Trees & XGBoost
   - Neural Networks 1-4
4. Reading Material & Preparation (see Absalon)
   - Large-Scale Least Squares: Check "Kernels", "Basic Kernel Methods", and maybe "SVMs"
   - Large-Scale Random Forests: Check "Trees"

Outline

Outline

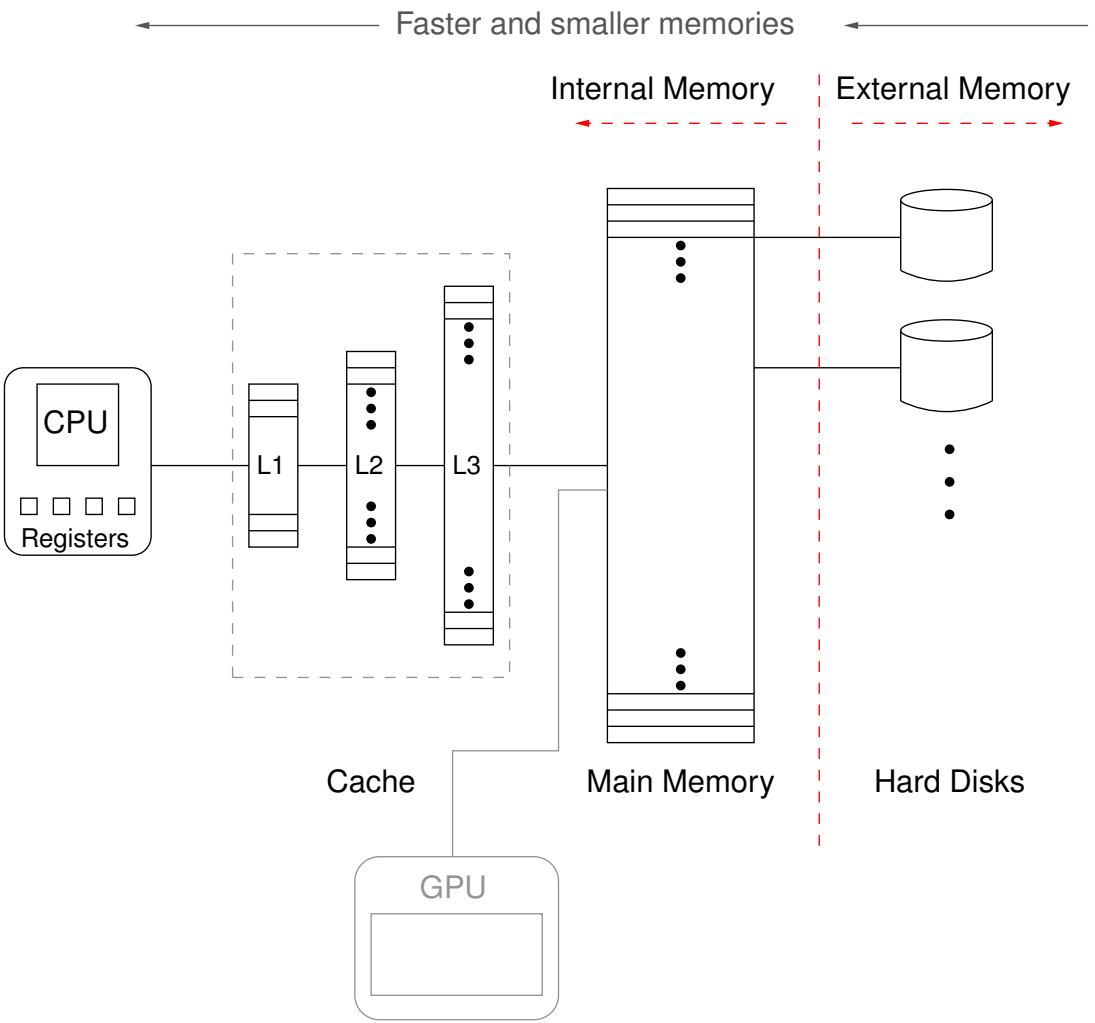
Computers I

Commodity Hardware

1. Usually: Single Central Processing Unit (CPU), 2-8 cores
2. Often: Graphics Processing Unit (GPU)
3. Main Memory: 4-64 GB
4. HDD/SSD: 128GB-6TB
5. ...

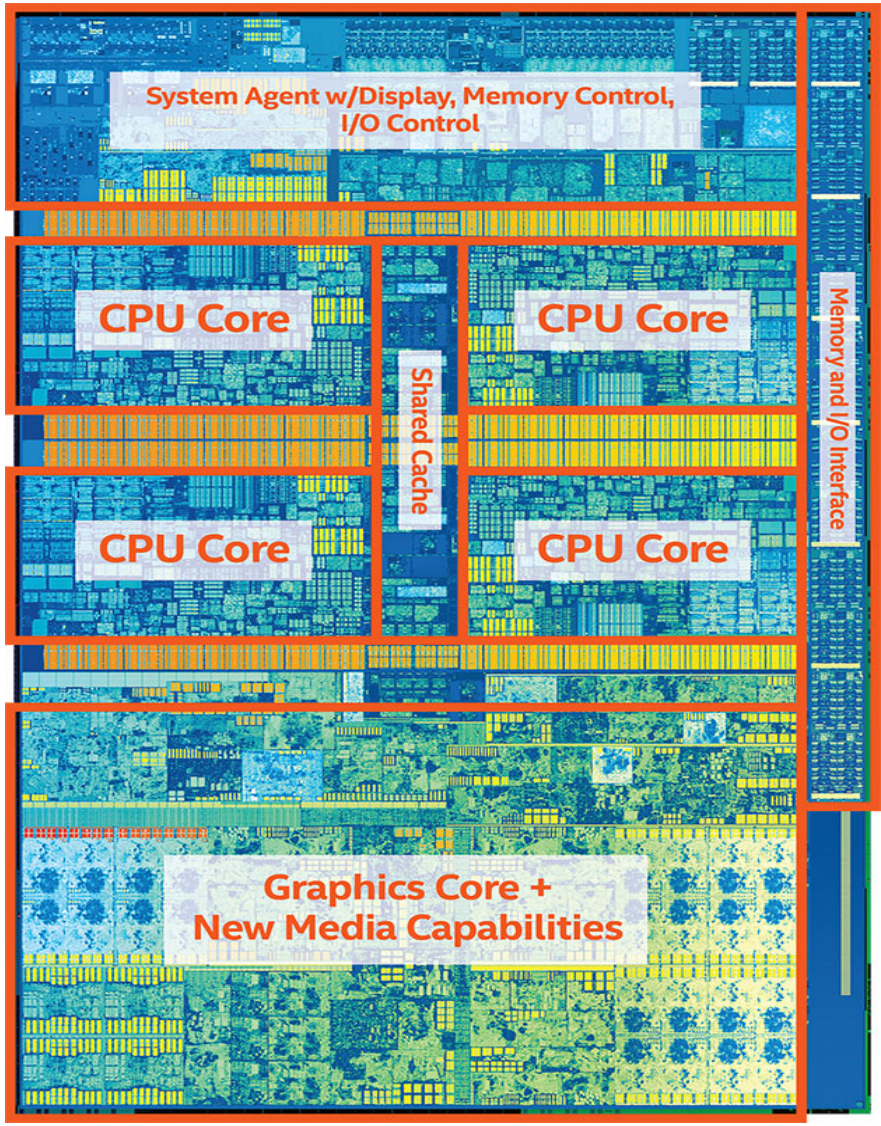
In a nutshell: Relatively cheap & efficient hardware, but often not very reliable ...

Computers II



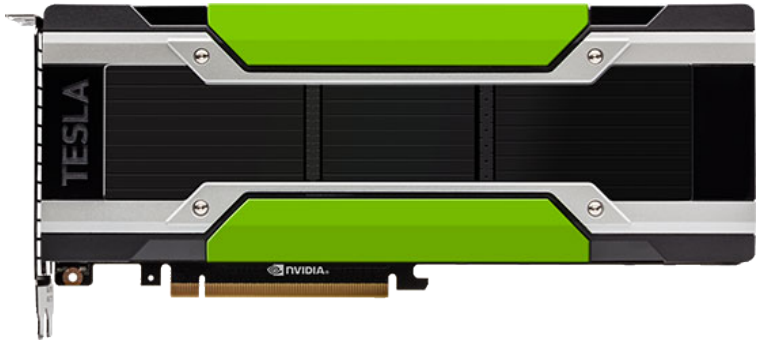
CPUs (2018)

- Multiple cores
  (2-16 for desktop CPUs)
- Multiple CPU sockets
  (usually 1-2 per board)
- Clock speed: 1.5 to 4.5 GHz
- Power consumption: 1 to 100W
- Fast caches L1, L2, and L3
  ($\approx$ 128KB, 1MB, and 8MB; bandwidths of up to 4TB/s)
- Interface to main memory
  DDR3 (6-17GB/s), DDR4 (12-20GB/s)
- Integrated low-end GPU unit
  (e.g., video decoding)
- GFLOPs: 50+

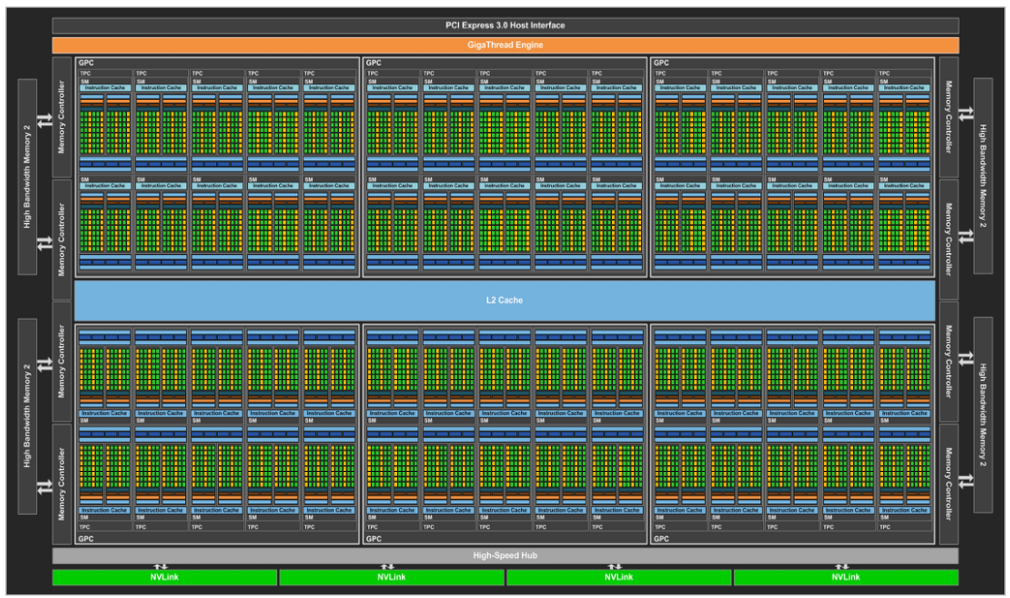


http://techhardware.com/reviews/intel-core-i7-7700k-and-z270-chipset-review

GPUs (2018)

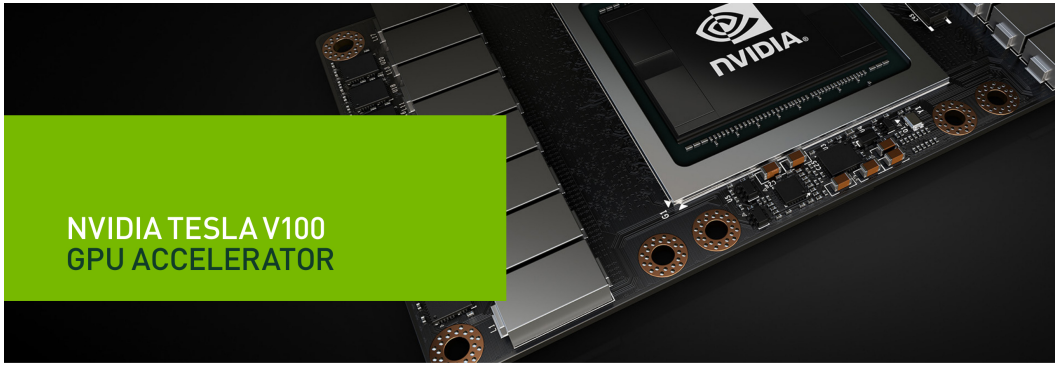


- Thousands of cores
  (e.g., P100: 3,584)
- Clock speed: $\approx$ 0.9 to 1.5GHz
- Implementing efficient GPU code is more difficult
  (e.g., cores execute instructions simultaneously, data access, . . .)
- Power consumption: 20-350W (e.g., 250W for P100)
- Fast caches L1, L2
  (e.g., L2 $\approx$ 4MB)
- Fast onboard memory
  (e.g., 16GB with 732GB/s)
- Host $\leftrightarrow$ GPU memory transfer
  (e.g., 12GB/s)
- GFLOPs: 5000+



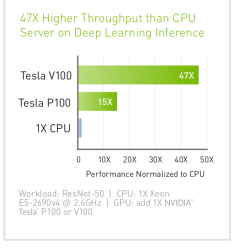
https://devblogs.nvidia.com/parallelforall/inside-pascal/
https://www.microway.com/hpc-tech-tips/nvidia-tesla-p100-nvlink-16gb-gpu-accelerator-pascal-gp100-sxm2-close/

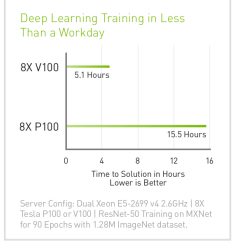
GPUs (2019)

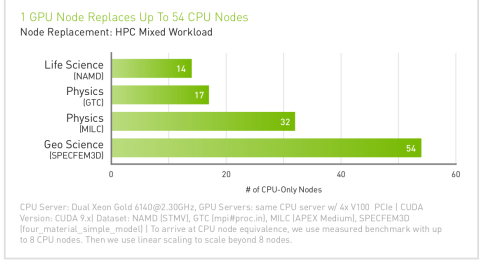


The Most Advanced Data Center GPU Ever Built.

NVIDIA® Tesla® V100 is the world's most advanced data center GPU ever built to accelerate AI, HPC, and graphics. Powered by NVIDIA Volta, the latest GPU architecture, Tesla V100 offers the performance of up to 100 CPUs in a single GPU—enabling data scientists, researchers, and engineers to tackle challenges that were once thought impossible.







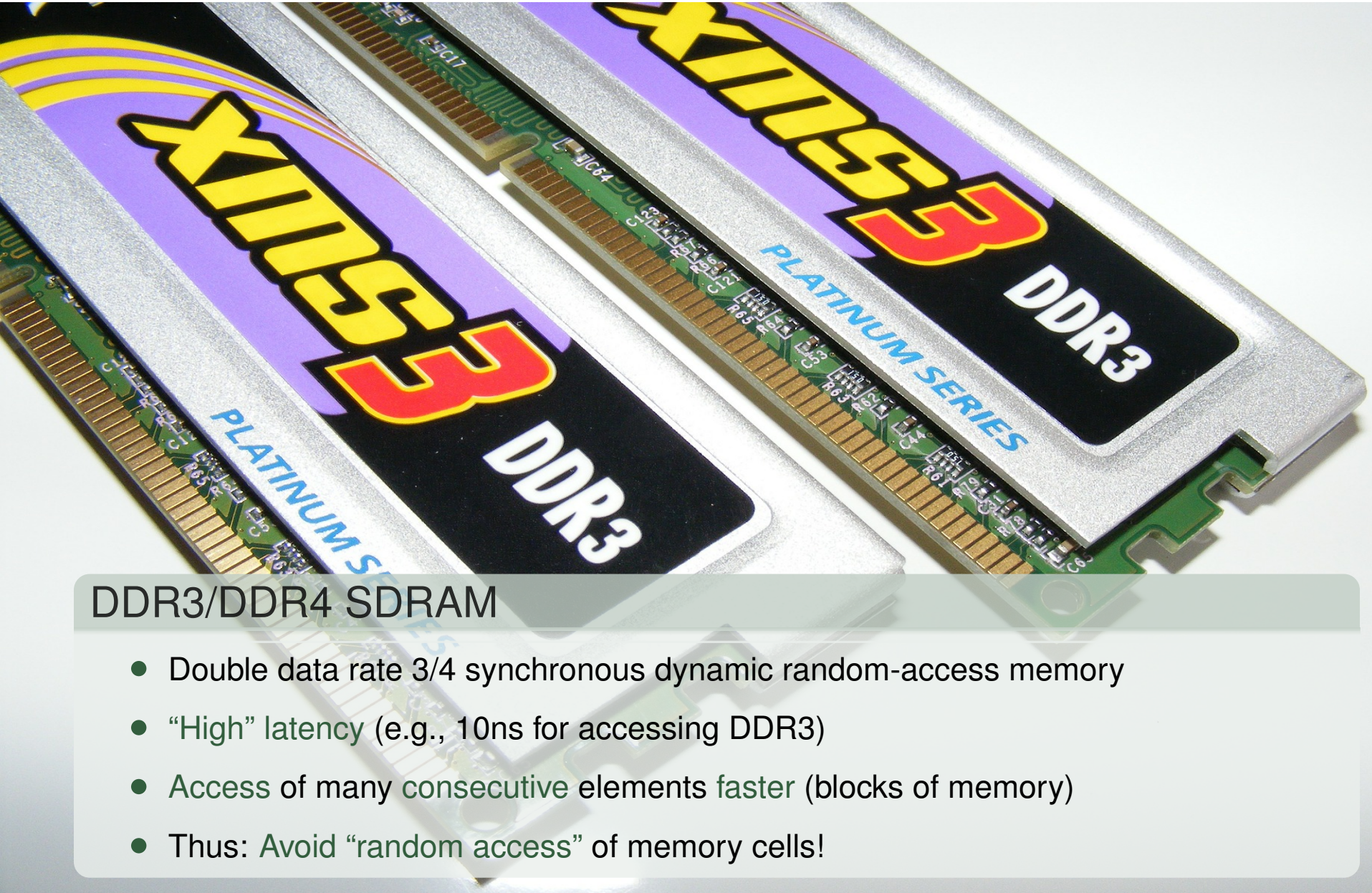
SPECIFICATIONS


	Tesla V100 PCIe	Tesla V100 SXM2
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
GPU Memory	32GB /16GB HBM2	
Memory Bandwidth	900GB/sec	
ECC	Yes	
Interconnect Bandwidth	32GB/sec	300GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink
Form Factor	PCIe Full Height/Length	SXM2
Max Power Comsumption	250 W	300 W
Thermal Solution	Passive	
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC	

TESLA V100 | DATA SHEET | MAR18

https://www.nvidia.com/en-us/data-center/tesla-v100/

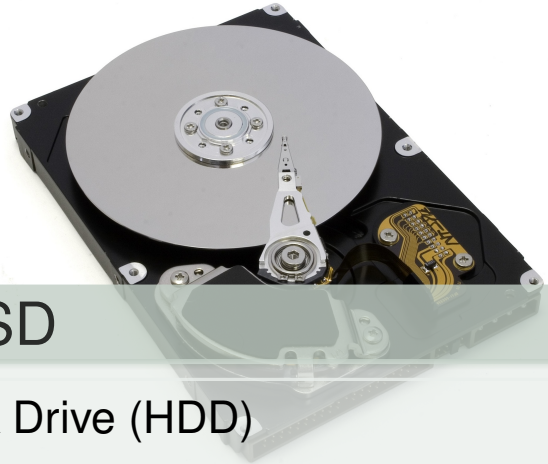
Internal Memory: RAM



DDR3/DDR4 SDRAM

- Double data rate 3/4 synchronous dynamic random-access memory
- “High” latency (e.g., 10ns for accessing DDR3)
- Access of many consecutive elements faster (blocks of memory)
- Thus: Avoid “random access” of memory cells!

External Memory: HDD, SSD, ...



HDD vs. SSD

1. Hard Disk Drive (HDD)
   - About 10TB of storage (0.05 €)
   - 200MB/s bandwidth (sequential read/writes)
   - Access time (seek/random access): 5-10 ms
2. Solid-State Drive (SSD)
   - About 4TB of storage (0.3 €)
   - 500MB/s bandwidth (sequential read/writes)
   - Access time (random access): $<$0.05 ms (x100 faster)
   - Reliable (no moving parts, but limited lifetime)

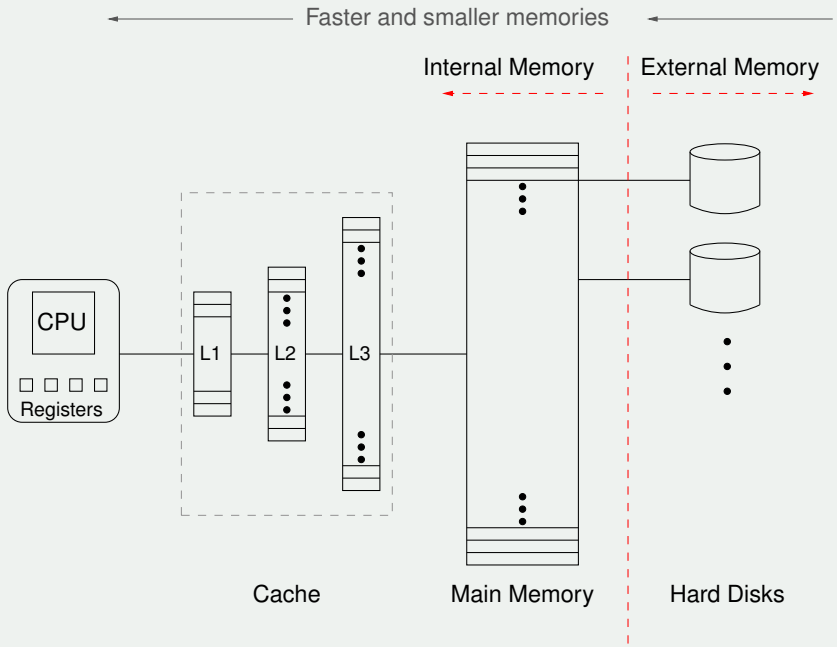
Thus: Similar bandwidth for consecutive memory access (e.g., loading a big file), but SSDs are much faster for many small random accesses (e.g., thousands of small files).

Accessing Data

Automatic Memory Transfers

Data can be loaded to main memory from external sources (and are indexed via memory addresses). Computations can only be done on items available in registers. In case an accessed data item is not in one of the registers, automatic memory transfers happen:

1. If data item is not in a register, it is fetched from L1 cache.
2. If data item is not in L1 cache, it is fetched from L2 cache.
3. If data item is not in L2 cache, it is fetched from L3 cache.
4. If data item is not in L3 cache, it is fetched from main memory.
5. Optional: Swap space



Transfer of data blocks: In case a data item is moved, a whole block of items is transferred! The idea is to benefit from temporal and spatial data locality.

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

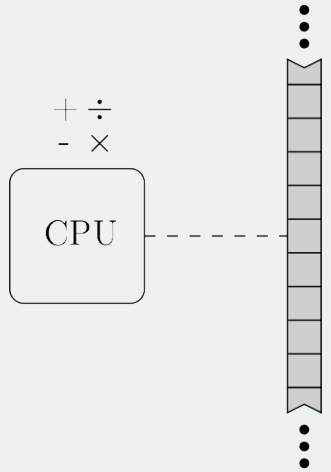
Google

https://research.google.com/people/jeff/stanford-295-talk.pdf

Abstraction (Computation Models): Example I

Real-RAM Model

1. Main Memory (unlimited number of cells).
2. CPU has direct access to memory.
3. Each cell can store a real number.
4. Complexity/runtime measured in terms of number of operations (each takes constant time):
   - Accessing a memory cell.
   - Arithmetic operations $(+, -, \times, \div)$.
   - Comparison of two real numbers $(<, \leq, =, \geq, >)$.
   - Indirect memory access (integer-based address).
   - $\sqrt{x}$, $\log x$, ...

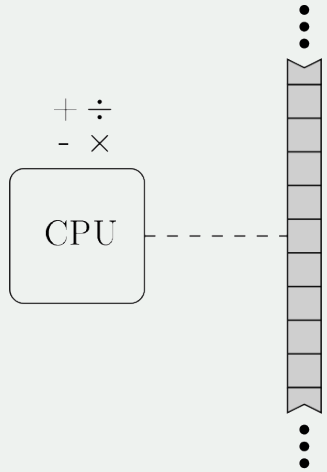


Preparata and Shamos: *Computational Geometry – An Introduction.* Springer, 1985.

Abstraction (Computation Models): Example I

Real-RAM Model

1. Main Memory (unlimited number of cells).
2. CPU has direct access to memory.
3. Each cell can store a real number.
4. Complexity/runtime measured in terms of number of operations (each takes constant time):
   - Accessing a memory cell.
   - Arithmetic operations $(+, -, \times, \div)$.
   - Comparison of two real numbers $(<, \leq, =, \geq, >)$.
   - Indirect memory access (integer-based address).
   - $\sqrt{x}$, $\log x$, ...



Preparata and Shamos: *Computational Geometry – An Introduction.* Springer, 1985.

Typically what people have in mind when talking about "runtimes" and "space complexity". One usually measures the asymptotic runtime/space complexity via the so-called big O-notation. For instance, "$O(n^3)$ time and $O(n^2)$ space".

Recap: Analyis of Algorithms

Asymptotic Runtimes

One is often interested in the asymptotic complexity of an algorithm (e.g., its runtime or its memory consumption).

$$
\begin{aligned}
O(g) &= \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists n_0 \in \mathbb{N}_0, \exists c > 0 : f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\} \\
\Omega(g) &= \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists n_0 \in \mathbb{N}_0, \exists c > 0 : f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0\} \\
\Theta(g) &= O(g) \cap \Omega(g)
\end{aligned}
$$

Often, one simply writes "$=$" instead of "$\in$" (e.g., $f = O(n^2)$).

Example: Sorting n elements (comparison-based)

1. Lower bound: $\Omega(n \log n)$
2. Optimal algorithms: Mergesort and Heapsort can sort n elements in $\Theta(n \log n)$ time in the worst case.
3. Often efficient in practice: Quicksort can sort n elements in $\Theta(n \log n)$ time in the average case.

Recap: Analyis of Algorithms

Asymptotic Runtimes

One is often interested in the asymptotic complexity of an algorithm (e.g., its runtime or its memory consumption).

$$
\begin{aligned}
O(g) &= \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists n_0 \in \mathbb{N}_0, \exists c > 0 : f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\} \\
\Omega(g) &= \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists n_0 \in \mathbb{N}_0, \exists c > 0 : f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0\} \\
\Theta(g) &= O(g) \cap \Omega(g)
\end{aligned}
$$

Often, one simply writes "$=$" instead of "$\in$" (e.g., $f = O(n^2)$).

Example: Sorting n elements (comparison-based)

1. Lower bound: $\Omega(n \log n)$
2. Optimal algorithms: Mergesort and Heapsort can sort n elements in $\Theta(n \log n)$ time in the worst case.
3. Often efficient in practice: Quicksort can sort n elements in $\Theta(n \log n)$ time in the average case.

Sometimes, multiple runtime parameters are considered at the same time. For instance: $O(d \cdot n)$

Quiz: Worst case upper runtime bounds for . . .

1. Computing the sum of n elements:
2. Binary search in sorted array with n elements:
3. Computation of $\mathbf{A} \cdot \mathbf{x}$ with $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^m$:
4. Computation of $\mathbf{A} \cdot \mathbf{B}$ with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$:
5. Computation of linear least-squares solution $(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, and $\lambda > 0$:

Quiz: Worst case upper runtime bounds for . . .

1. Computing the sum of n elements:
   $O(n)$ time
2. Binary search in sorted array with n elements:
3. Computation of $\mathbf{A} \cdot \mathbf{x}$ with $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^m$:
4. Computation of $\mathbf{A} \cdot \mathbf{B}$ with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$:
5. Computation of linear least-squares solution $(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, and $\lambda > 0$:

Quiz: Worst case upper runtime bounds for . . .

1. Computing the sum of n elements:
   $\mathcal{O}(n)$ time
2. Binary search in sorted array with n elements:
   $\mathcal{O}(\log n)$ time
3. Computation of $\mathbf{A} \cdot \mathbf{x}$ with $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^m$:
4. Computation of $\mathbf{A} \cdot \mathbf{B}$ with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$:
5. Computation of linear least-squares solution $(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, and $\lambda > 0$:

Quiz: Worst case upper runtime bounds for . . .

1. Computing the sum of n elements:
   $\mathcal{O}(n)$ time
2. Binary search in sorted array with n elements:
   $\mathcal{O}(\log n)$ time
3. Computation of $\mathbf{A} \cdot \mathbf{x}$ with $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^m$:
   $\mathcal{O}(n \cdot m)$ time
4. Computation of $\mathbf{A} \cdot \mathbf{B}$ with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$:
5. Computation of linear least-squares solution $(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, and $\lambda > 0$:

Quiz: Worst case upper runtime bounds for . . .

1. Computing the sum of n elements:
   $\mathcal{O}(n)$ time
2. Binary search in sorted array with n elements:
   $\mathcal{O}(\log n)$ time
3. Computation of $\mathbf{A} \cdot \mathbf{x}$ with $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^m$:
   $\mathcal{O}(n \cdot m)$ time
4. Computation of $\mathbf{A} \cdot \mathbf{B}$ with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$:
   Direct implementation: $\mathcal{O}(n^3)$ time
   Le Gall (2014): $\mathcal{O}(n^{2.3728639})$
5. Computation of linear least-squares solution $(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, and $\lambda > 0$:

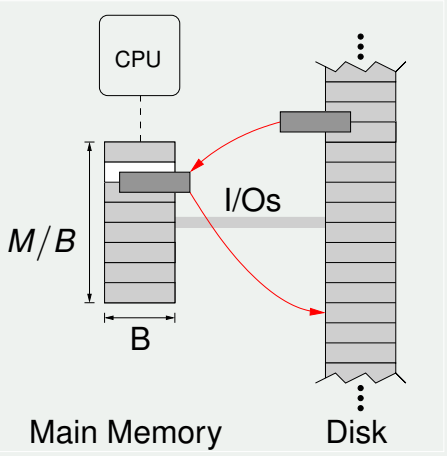
Quiz: Worst case upper runtime bounds for . . .

1. Computing the sum of n elements:
   $O(n)$ time
2. Binary search in sorted array with n elements:
   $O(\log n)$ time
3. Computation of $\mathbf{A} \cdot \mathbf{x}$ with $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^m$:
   $O(n \cdot m)$ time
4. Computation of $\mathbf{A} \cdot \mathbf{B}$ with $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$:
   Direct implementation: $O(n^3)$ time
   Le Gall (2014): $O(n^{2.3728639})$
5. Computation of linear least-squares solution $(\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, and $\lambda > 0$:
   $O(d^2 \cdot n + d + d^3 + d \cdot n + d^2) = O(d^2 \cdot n + d^3)$ time

Abstraction (Computation Models): Example II

Two-level I/O-Model

1. Main memory computations for free.
2. Complexity is measured in terms of I/Os.
3. N = problem size
4. M = memory size
5. B = block size



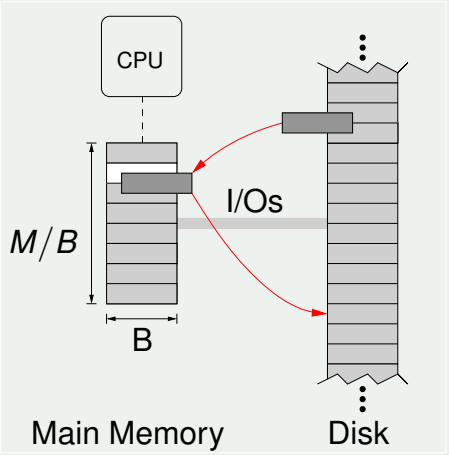
Alok Aggarwal and Jeffrey Scott Vitter: *The Input/Output Complexity of Sorting and Related Problems.* Communications of the ACM, 31(9):1116–1127, 1988.

Key idea: Measure performance of an algorithm in terms of memory transfers between main memory and (slow) disk. Avoid “swapping”!

Abstraction (Computation Models): Example II

Two-level I/O-Model

1. Main memory computations for free.
2. Complexity is measured in terms of I/Os.
3. N = problem size
4. M = memory size
5. B = block size



Alok Aggarwal and Jeffrey Scott Vitter: *The Input/Output Complexity of Sorting and Related Problems.* Communications of the ACM, 31(9):1116–1127, 1988.

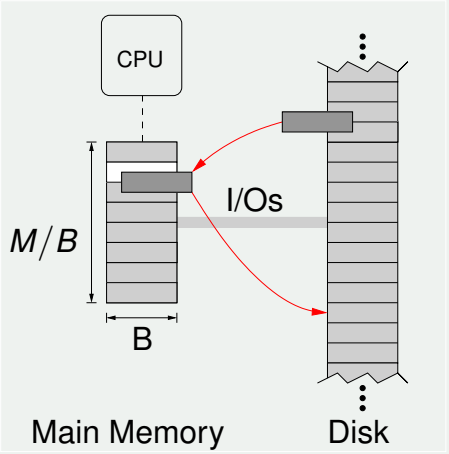
Key idea: Measure performance of an algorithm in terms of memory transfers between main memory and (slow) disk. Avoid “swapping”!

Question: What is the complexity of scanning N items?

Abstraction (Computation Models): Example II

Two-level I/O-Model

1. Main memory computations for free.
2. Complexity is measured in terms of I/Os.
3. N = problem size
4. M = memory size
5. B = block size

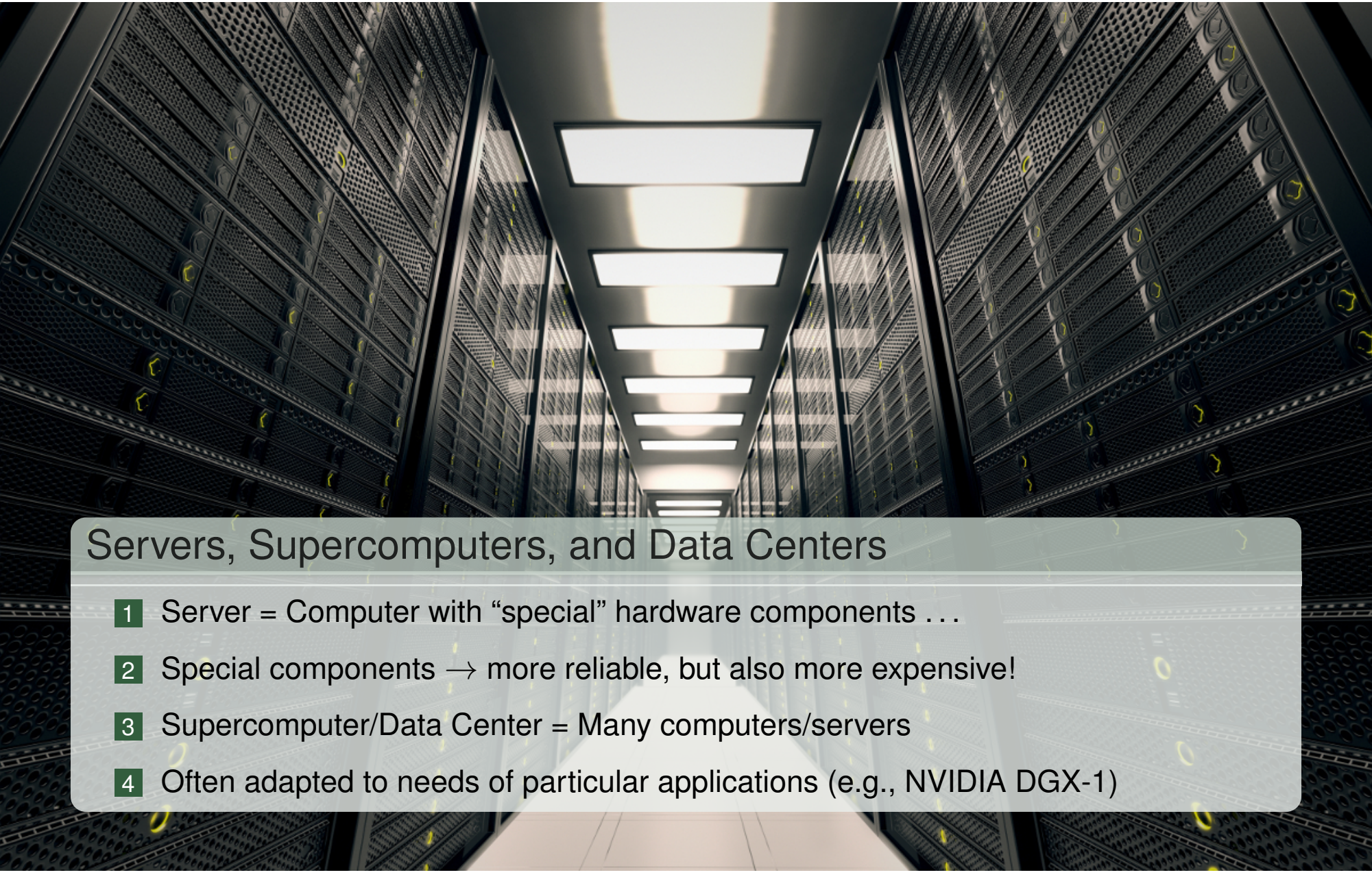


Alok Aggarwal and Jeffrey Scott Vitter: *The Input/Output Complexity of Sorting and Related Problems.* Communications of the ACM, 31(9):1116–1127, 1988.

Key idea: Measure performance of an algorithm in terms of memory transfers between main memory and (slow) disk. Avoid “swapping”!

- Scan items in $\mathcal{O}(\frac{N}{B})$ memory transfers
- Sort items in $\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$ memory transfers (almost scanning complexity!)

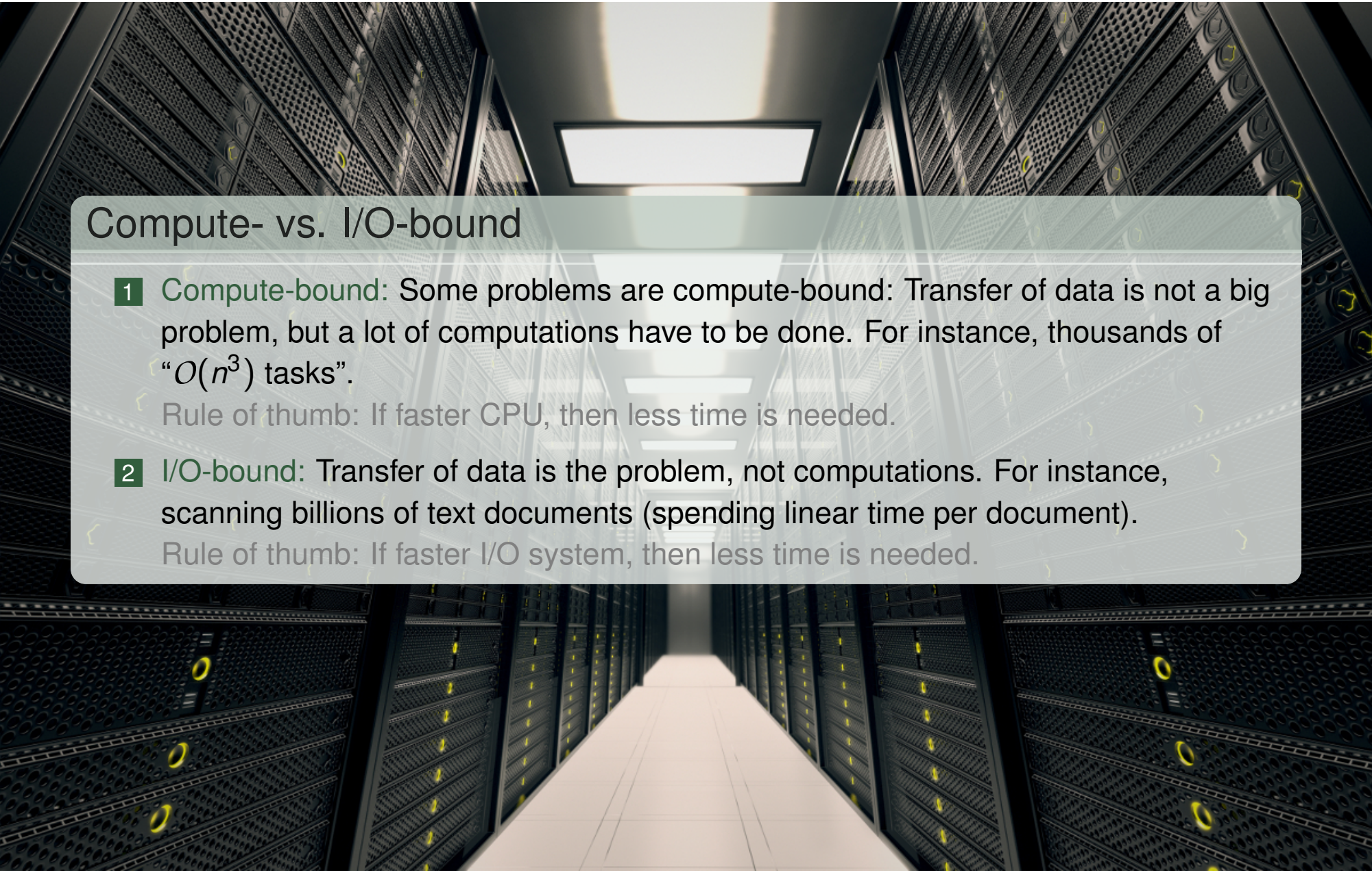
Big Computers



Servers, Supercomputers, and Data Centers

1. Server = Computer with “special” hardware components . . .
2. Special components $\rightarrow$ more reliable, but also more expensive!
3. Supercomputer/Data Center = Many computers/servers
4. Often adapted to needs of particular applications (e.g., NVIDIA DGX-1)

Using (Super-)Computers for Big Data

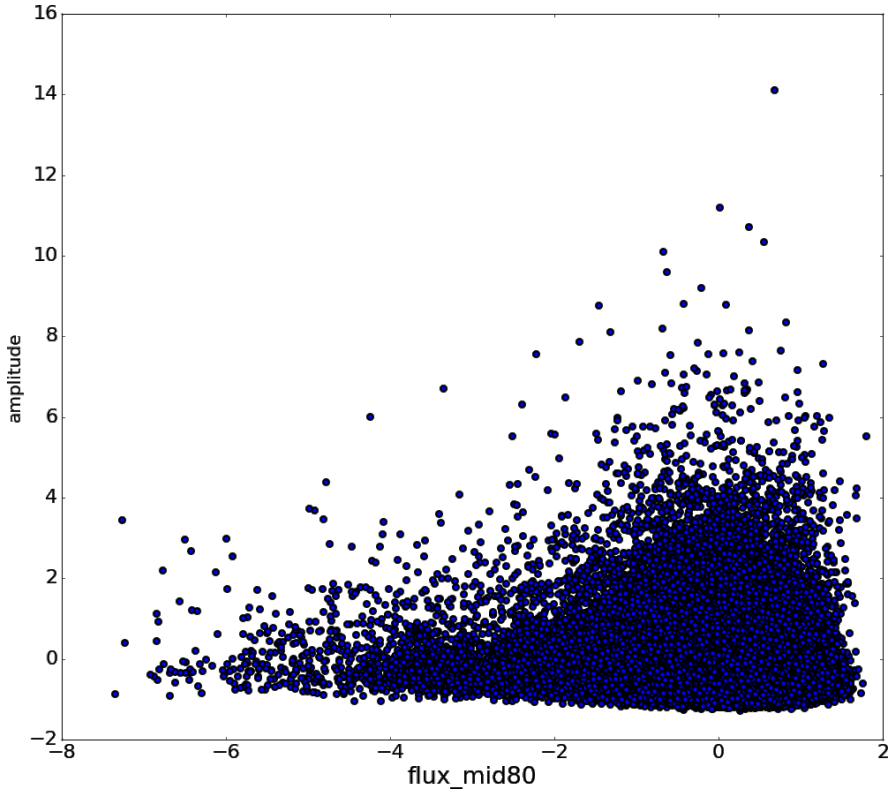


Compute- vs. I/O-bound

1. Compute-bound: Some problems are compute-bound: Transfer of data is not a big problem, but a lot of computations have to be done. For instance, thousands of "$O(n^3)$ tasks".
   Rule of thumb: If faster CPU, then less time is needed.
2. I/O-bound: Transfer of data is the problem, not computations. For instance, scanning billions of text documents (spending linear time per document).
   Rule of thumb: If faster I/O system, then less time is needed.

Outline

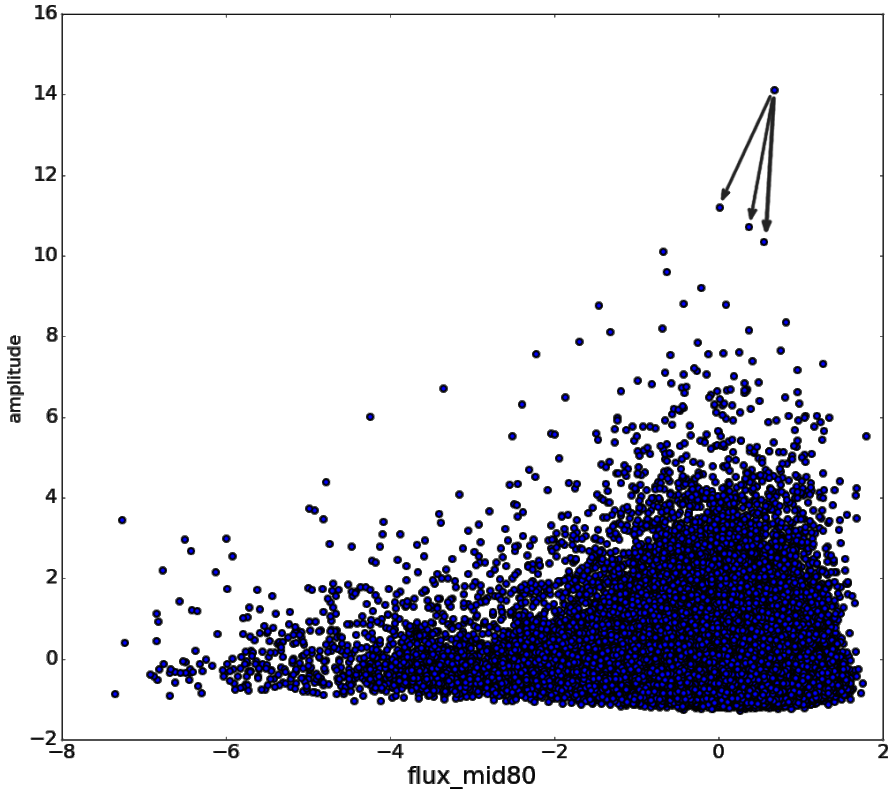
Example I: Outlier Detection



Outlier Detection

"I have one billion objects and each of them is described via 10 values (features). Can you find the outliers for me, i.e., objects that are somehow different from the other ones?"

Example I: Outlier Detection

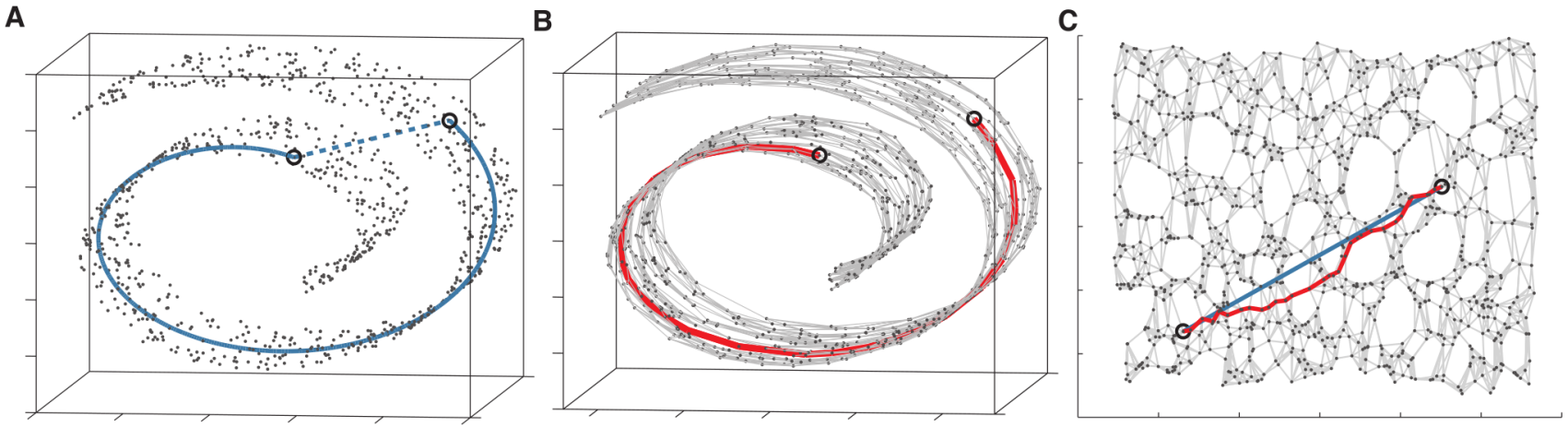


Outlier Detection

"I have one billion objects and each of them is described via 10 values (features). Can you find the outliers for me, i.e., objects that are somehow different from the other ones?"

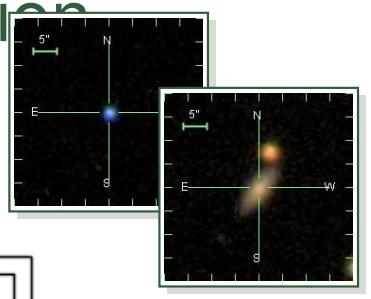
→ Compute, for each point, the distance to its nearest neighbors!

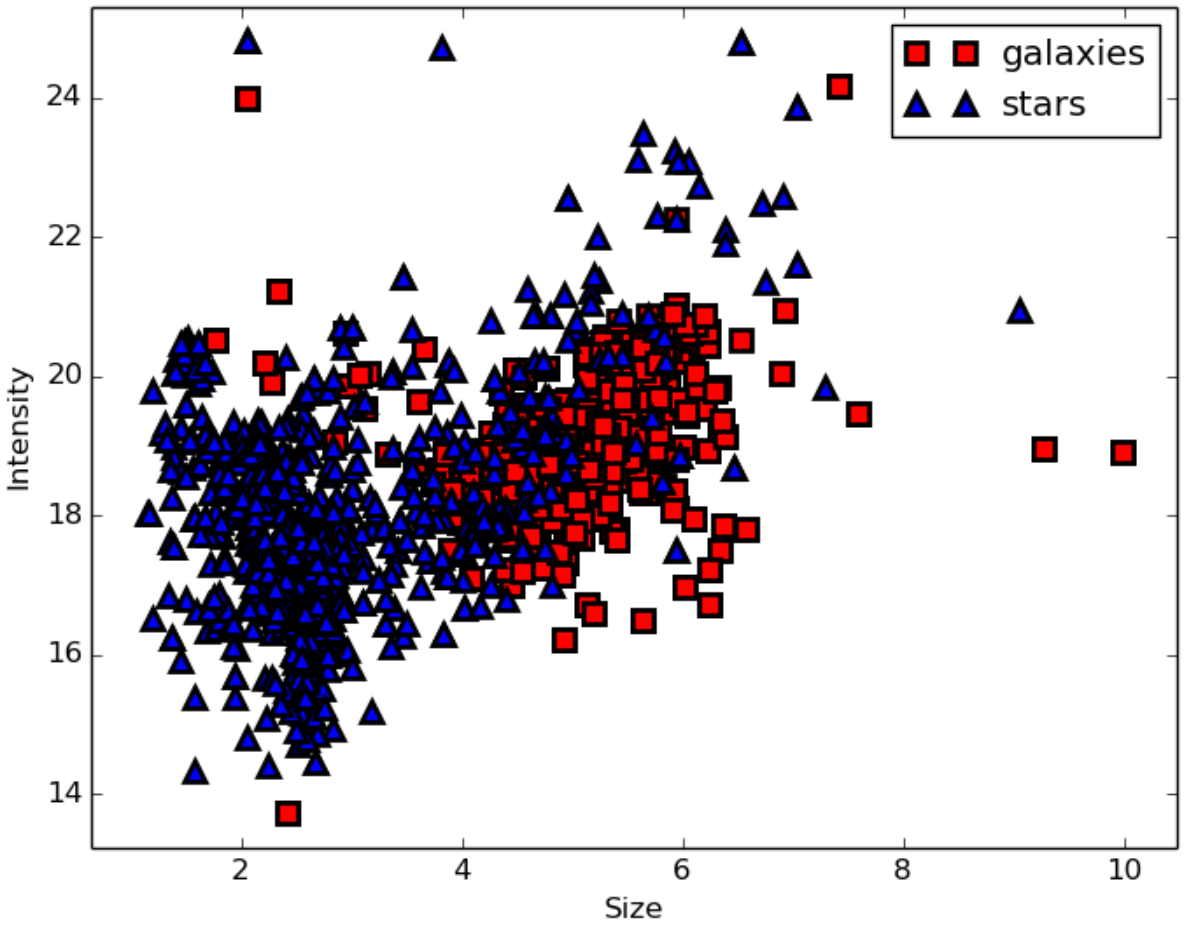
Example II: ISOMAP



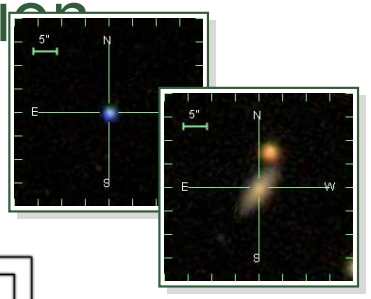
A Global Geometric Framework for Nonlinear Dimensionality Reduction.
Tenenbaum, Silva, Langford. Science, 290(5500), 2319–2323, 2000.

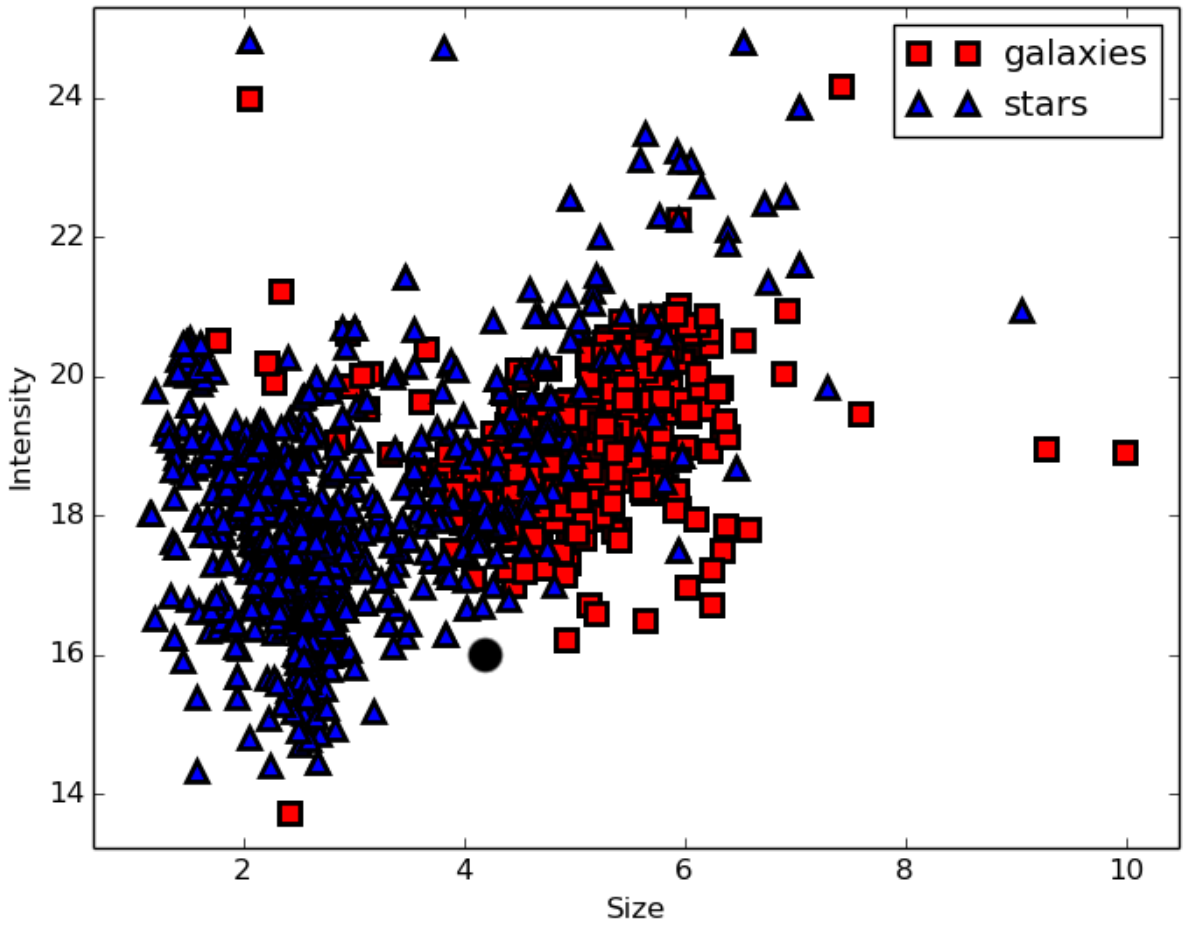
Example III: Nearest Neighbor Classification



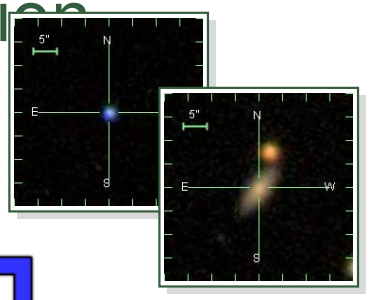


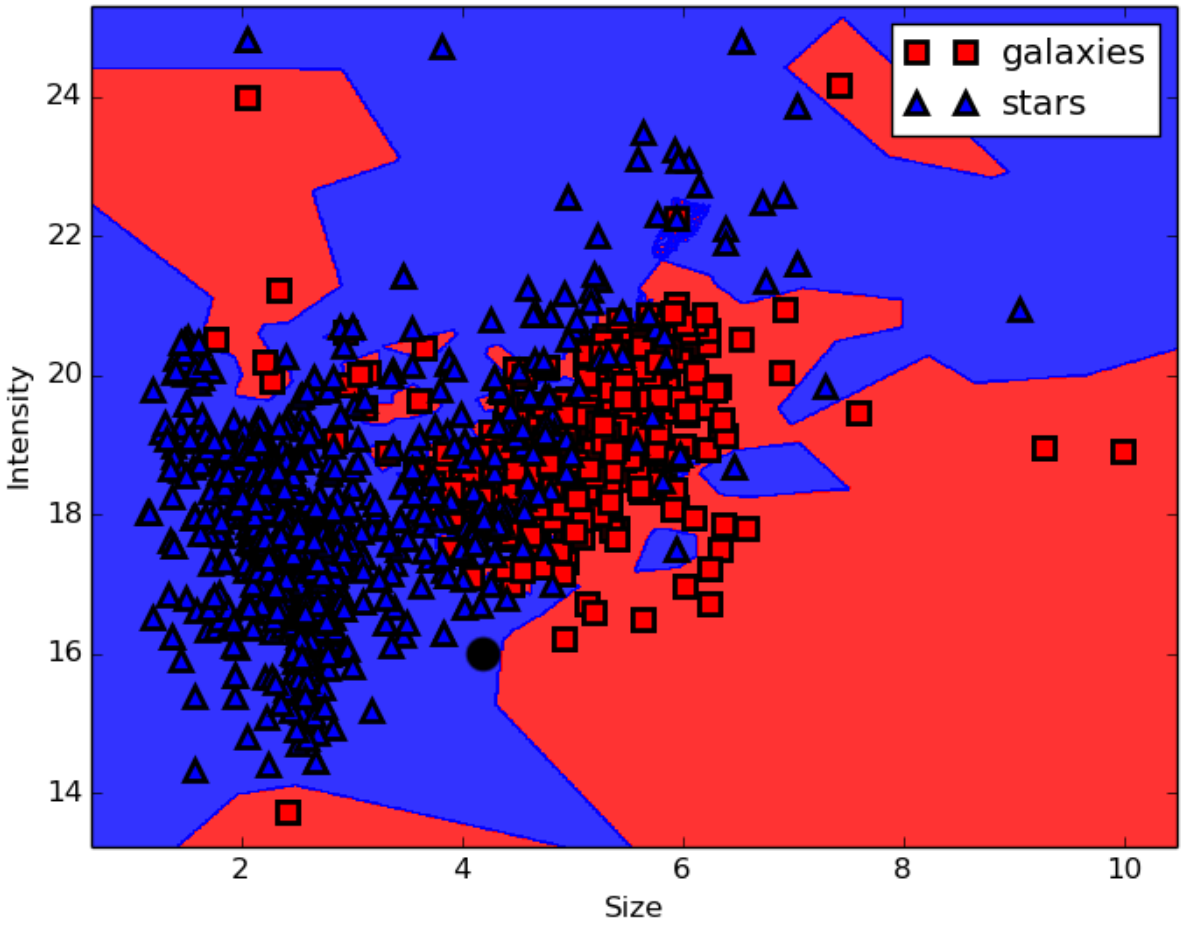
Example III: Nearest Neighbor Classification



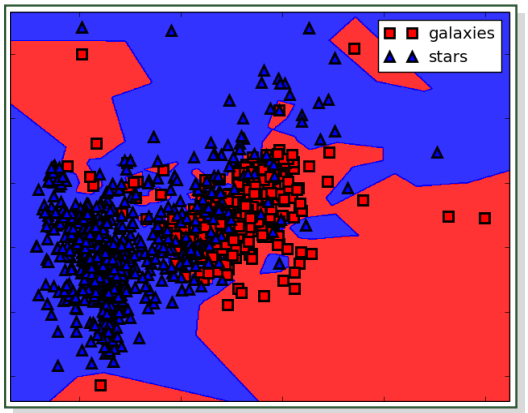


Example III: Nearest Neighbor Classification

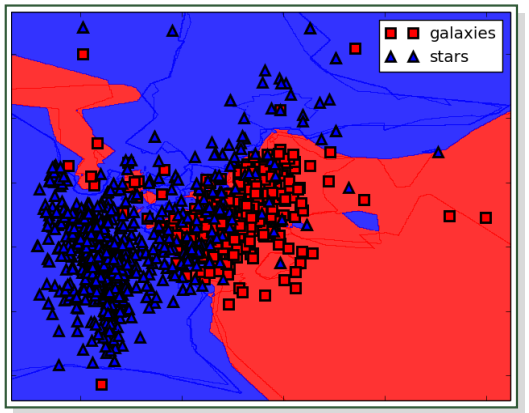




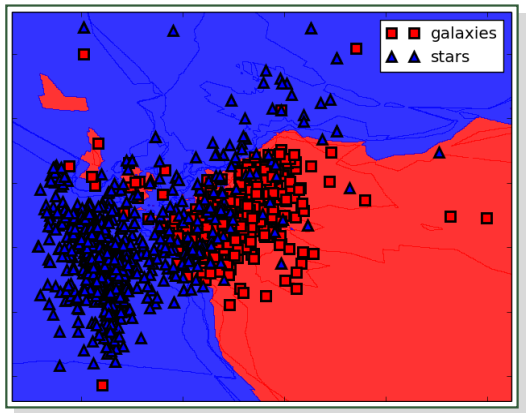
Recap: Nearest Neighbor Classification



$k = 1$



$k = 5$



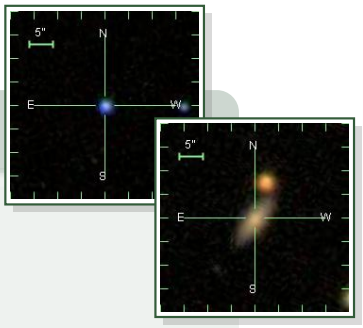
$k = 10$

The k-nearest neighbor classification algorithm

Require: Let $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times \{1, \ldots, C\}$ be the set of training examples and k be the number of nearest neighbors.

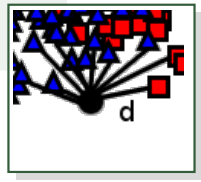
1: **for** each test example **x do**
2: Compute the distance $D(\mathbf{x}, \mathbf{x}_i)$ to each training example $\mathbf{x}_i$.
3: Select $N_{\mathbf{x}} \subset T$, the set of the k closest training examples to $\mathbf{x}$.
4: $f(\mathbf{x}) = \text{argmax}_c \sum_{(\mathbf{x}_i, y_i) \in N_{\mathbf{x}}} \mathbb{I}(c = y_i)$
5: **end for**

Nearest Neighbor Search



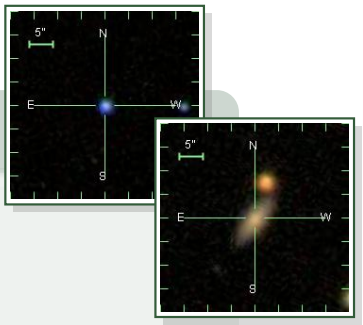
Brute-Force Approach

```
1  for i in range(len(train)):
2      for j in range(len(test)):
3          d = distance(train[i], test[q])
4          update_neighbors(d, i, j)
5          ...
```

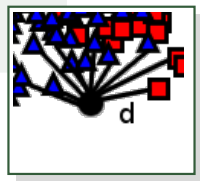


Nearest Neighbor Search

Brute-Force Approach




```
1  for i in range(len(train)):
2      for j in range(len(test)):
3          d = distance(train[i], test[q])
4          update_neighbors(d, i, j)
5          ...
```

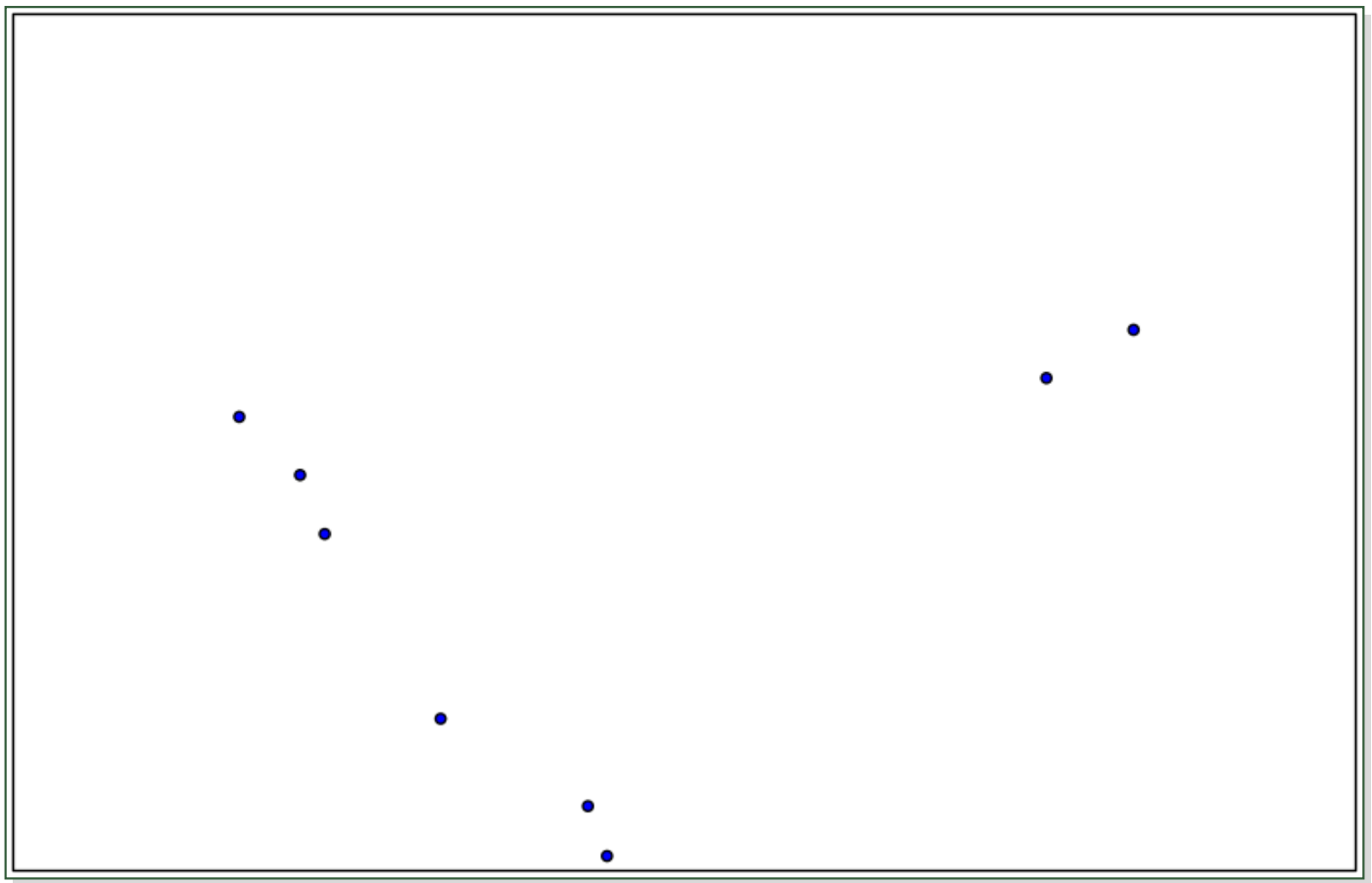


Very broad and active research field. Typical techniques are:

1. Parallel implementations (e.g., brute-force + GPUs)
2. Spatial search structures (e.g., k-d trees or ball trees)
3. Approximation schemes (e.g., locality-sensitive hashing)
4. Compression/reduction (e.g., reduce training set in a “clever” way)
5. ...

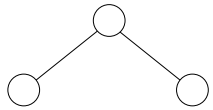
In the following: Spatial search structures, approximation schemes, parallel implementations, ...

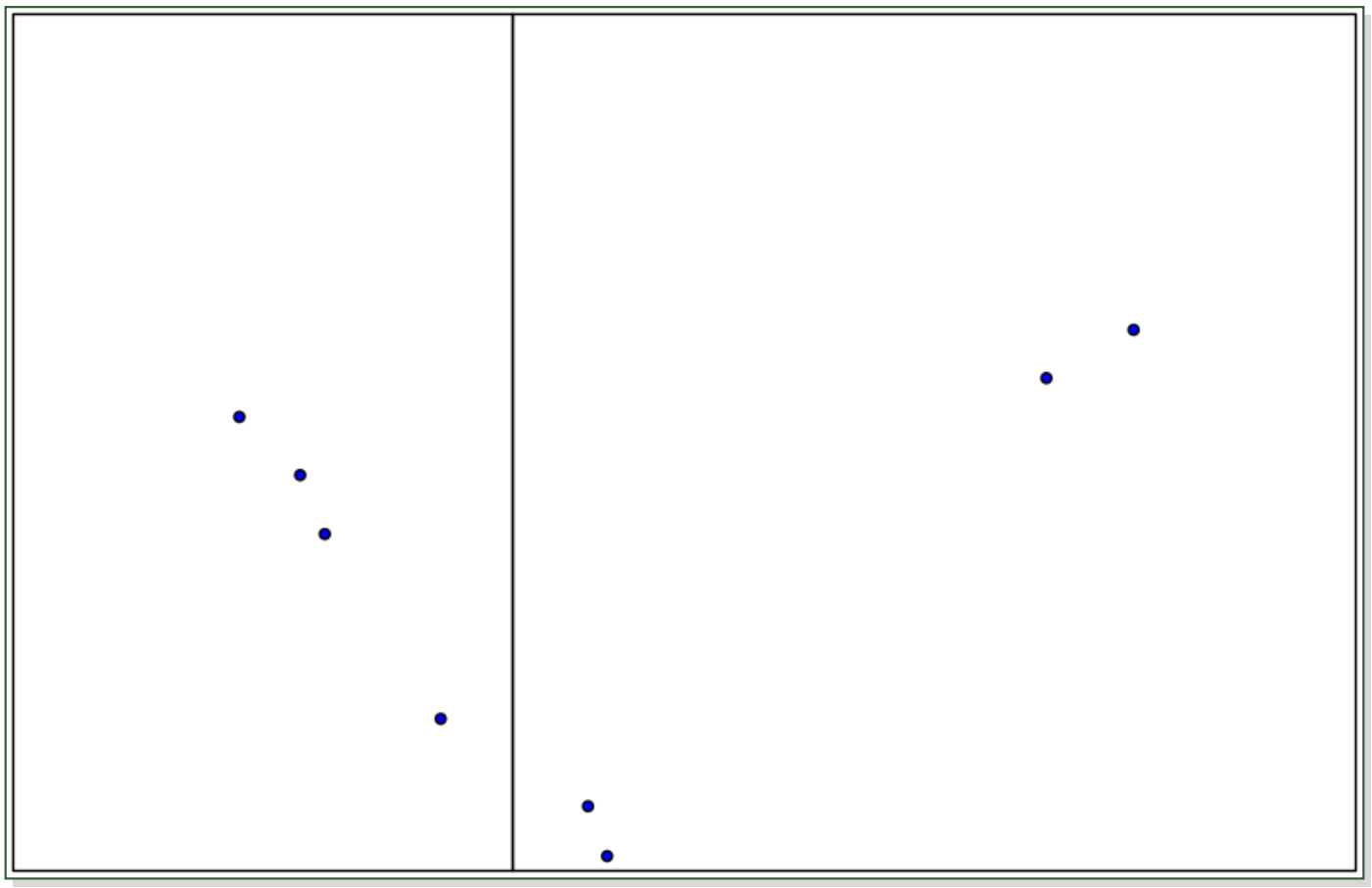
A Classic: k-d Trees



Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

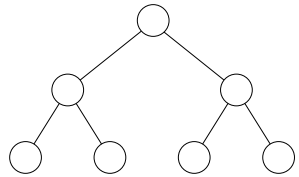
A Classic: *k*-d Trees

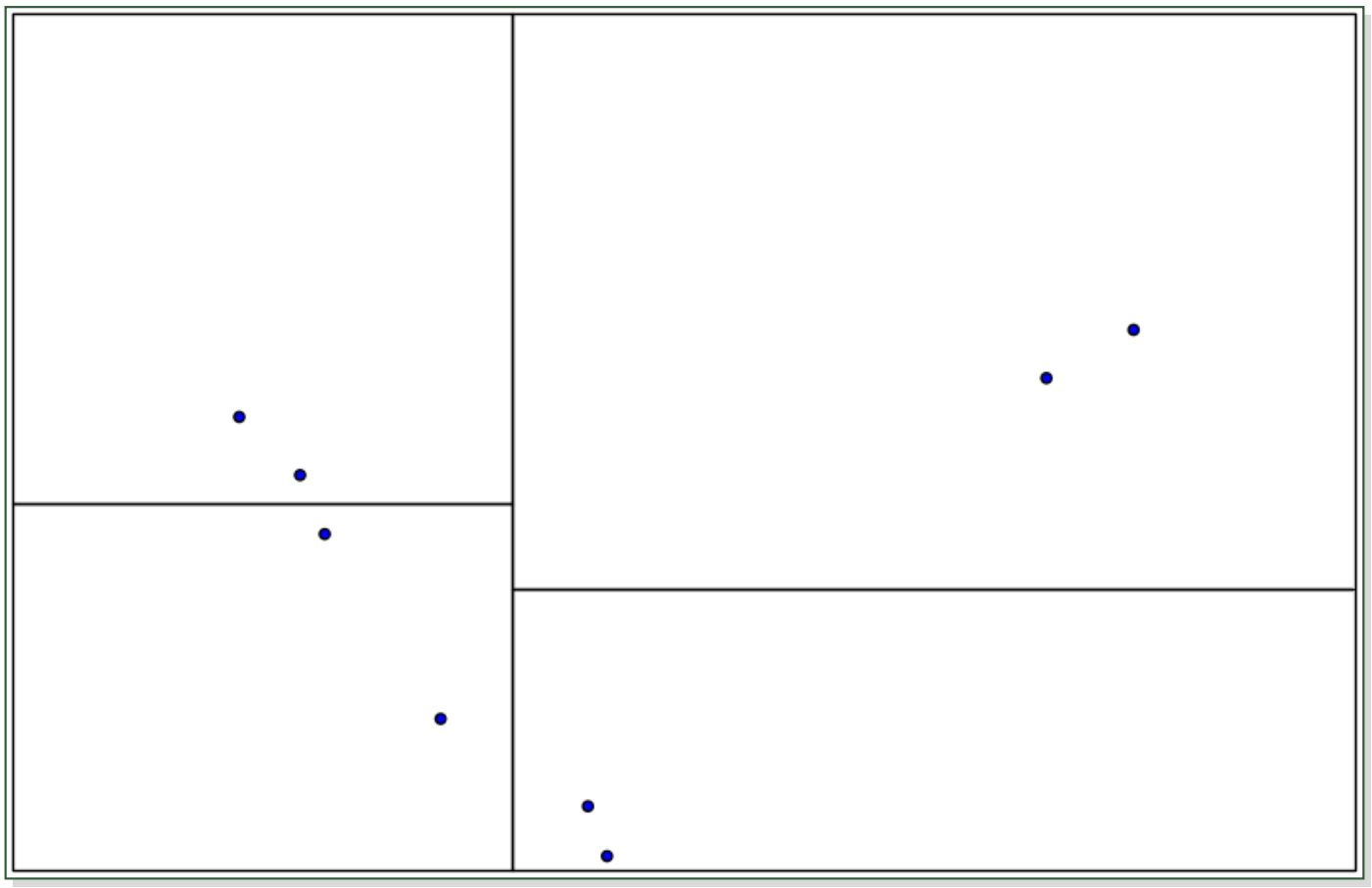




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

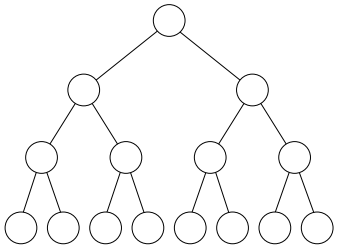
A Classic: *k*-d Trees

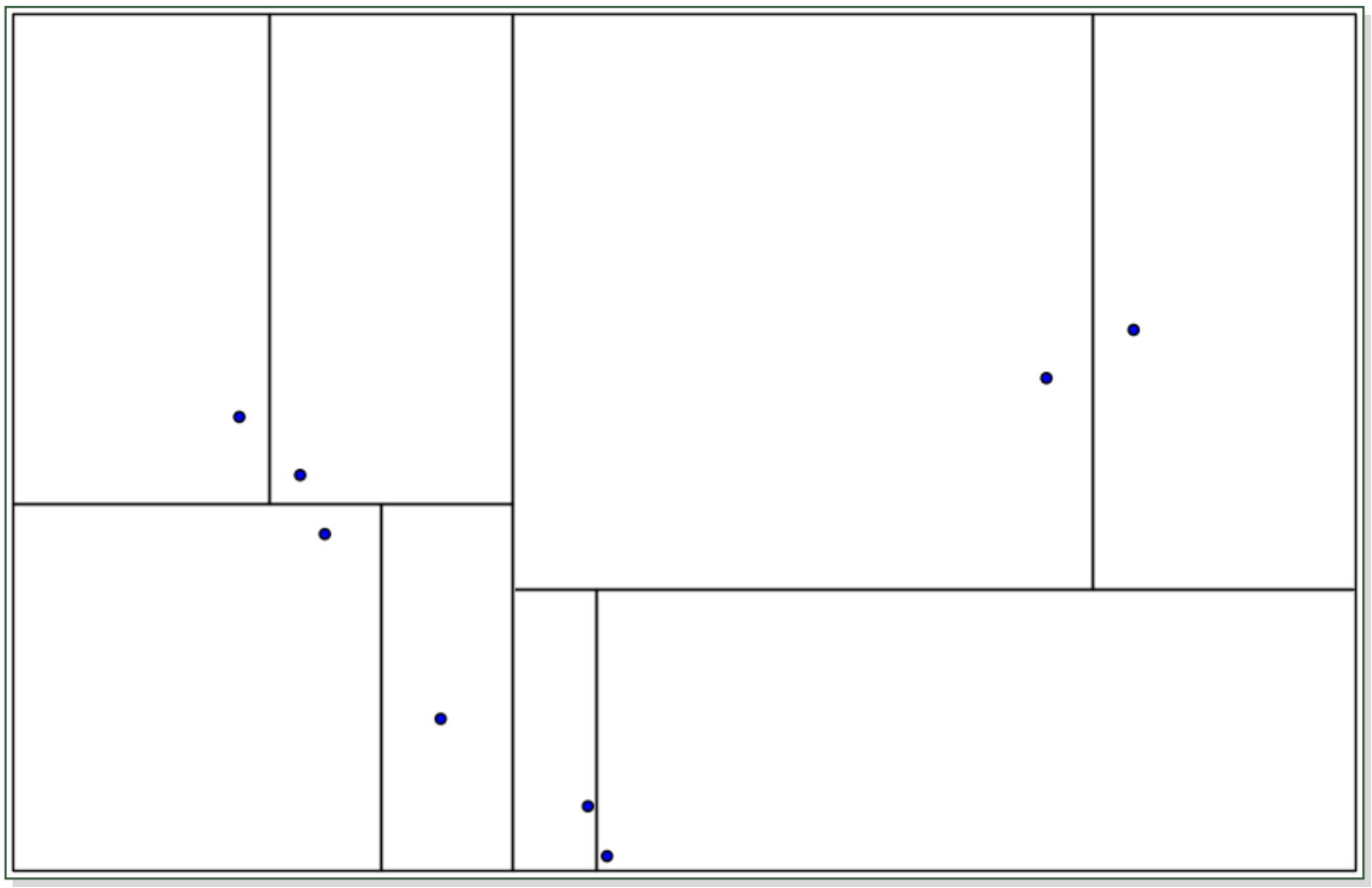




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

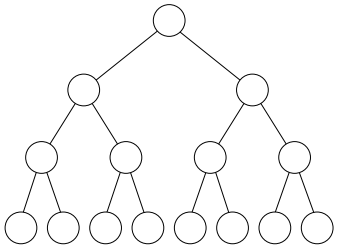
A Classic: *k*-d Trees

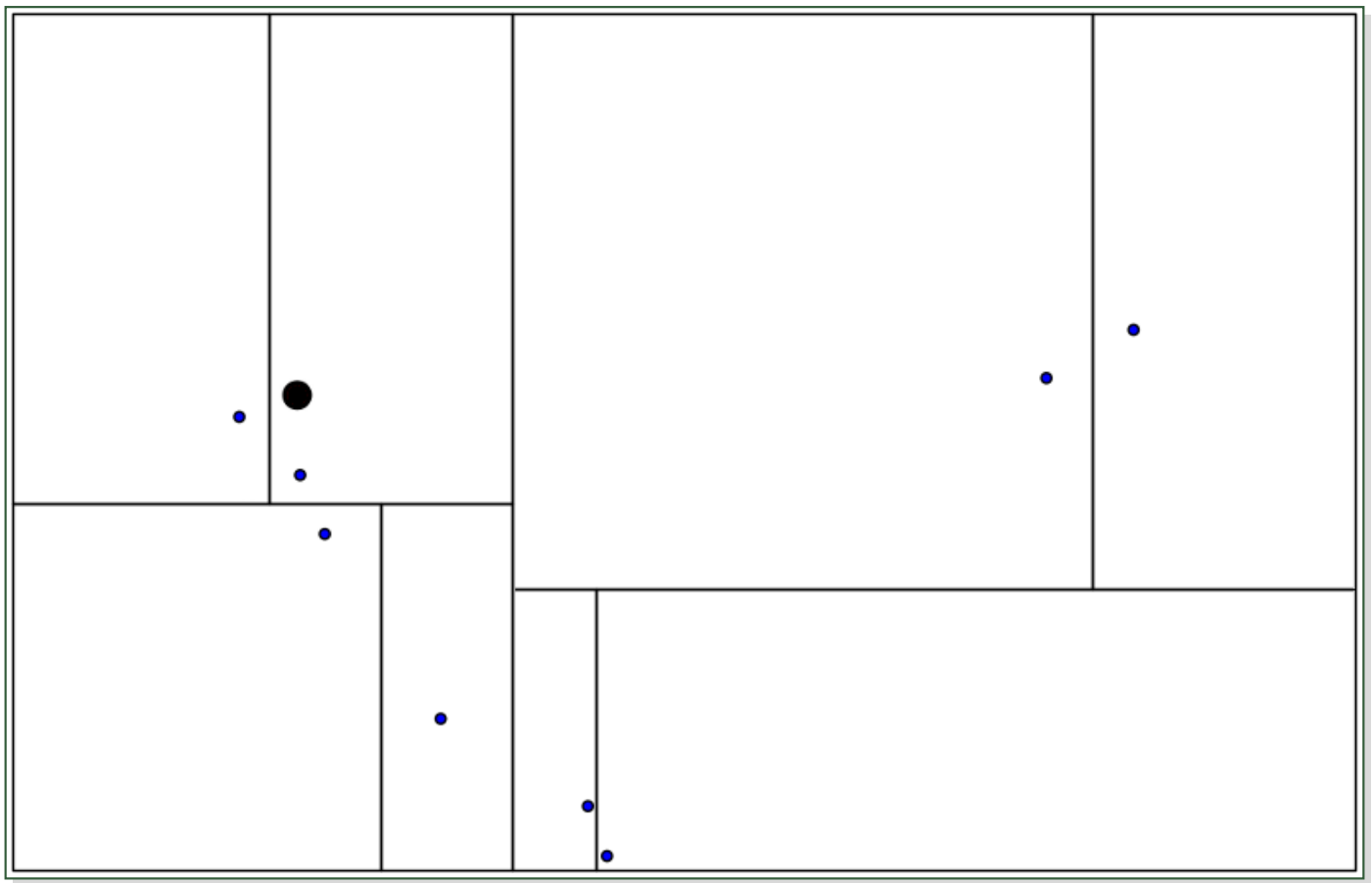




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

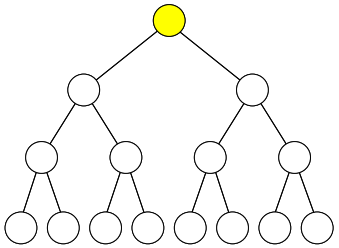
A Classic: *k*-d Trees

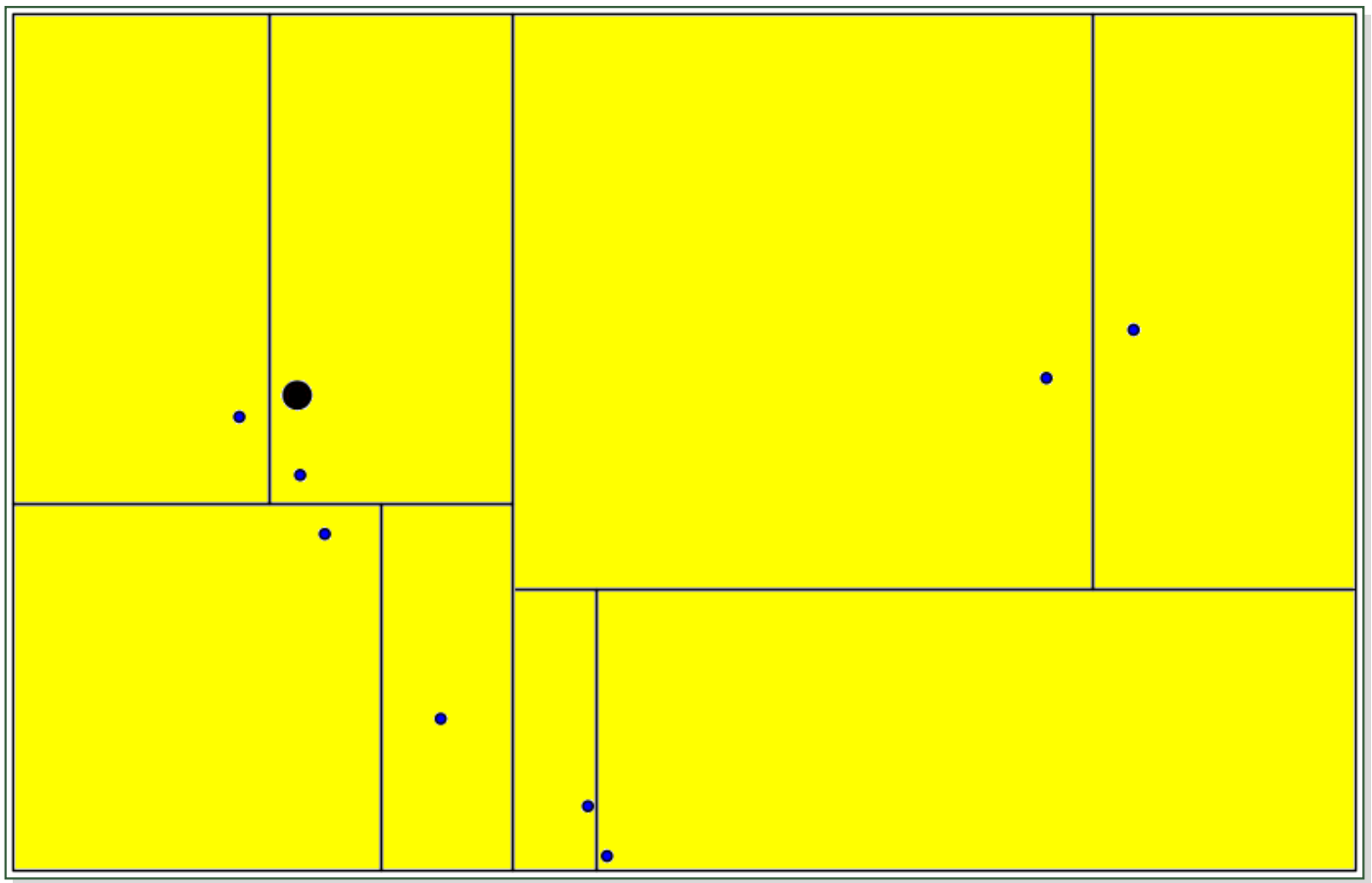




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

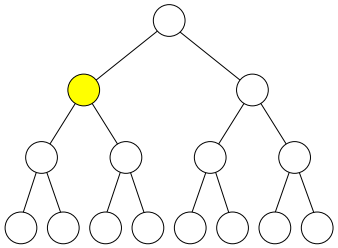
A Classic: *k*-d Trees

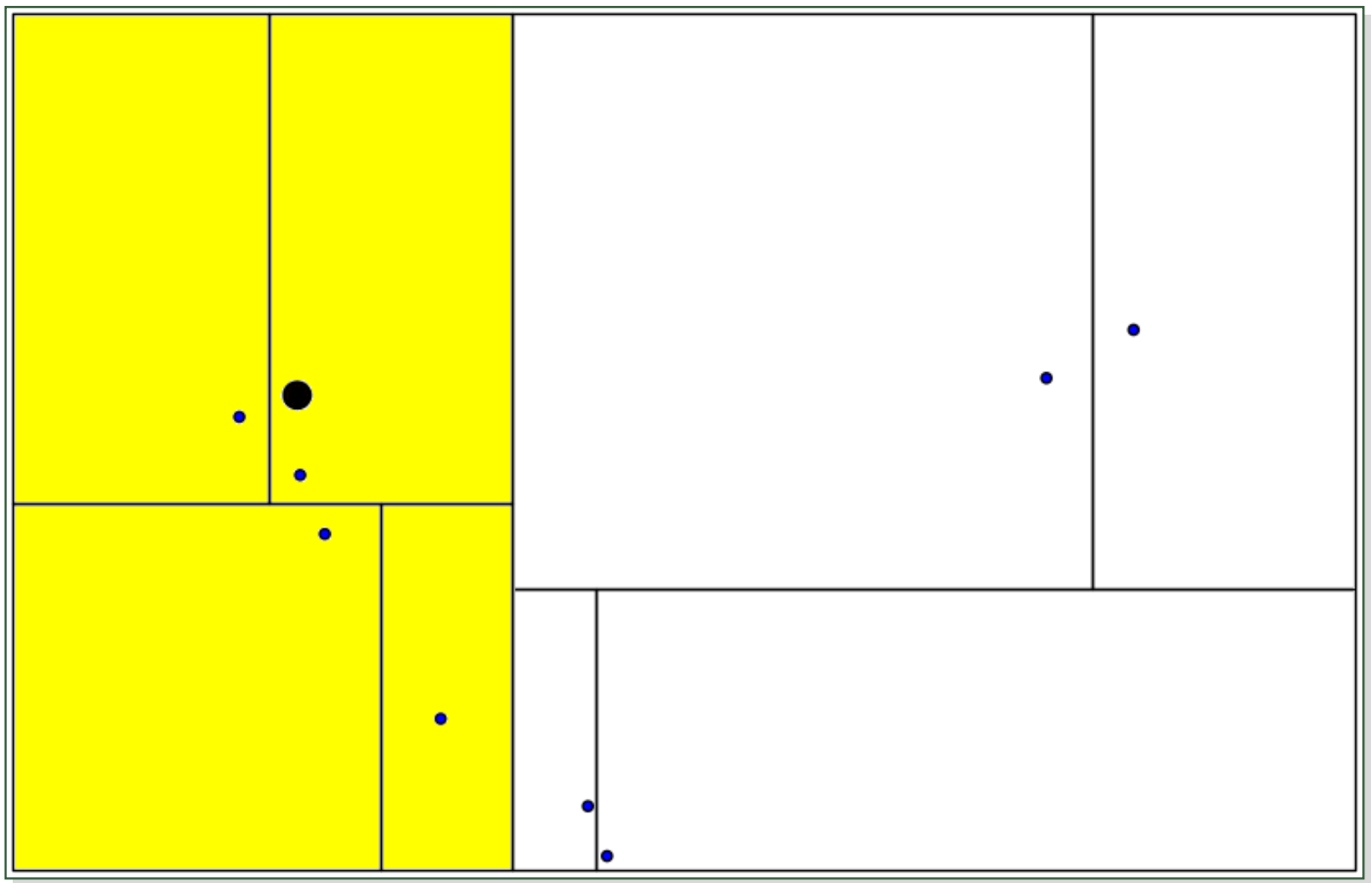




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

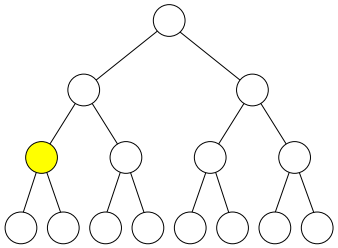
A Classic: *k*-d Trees

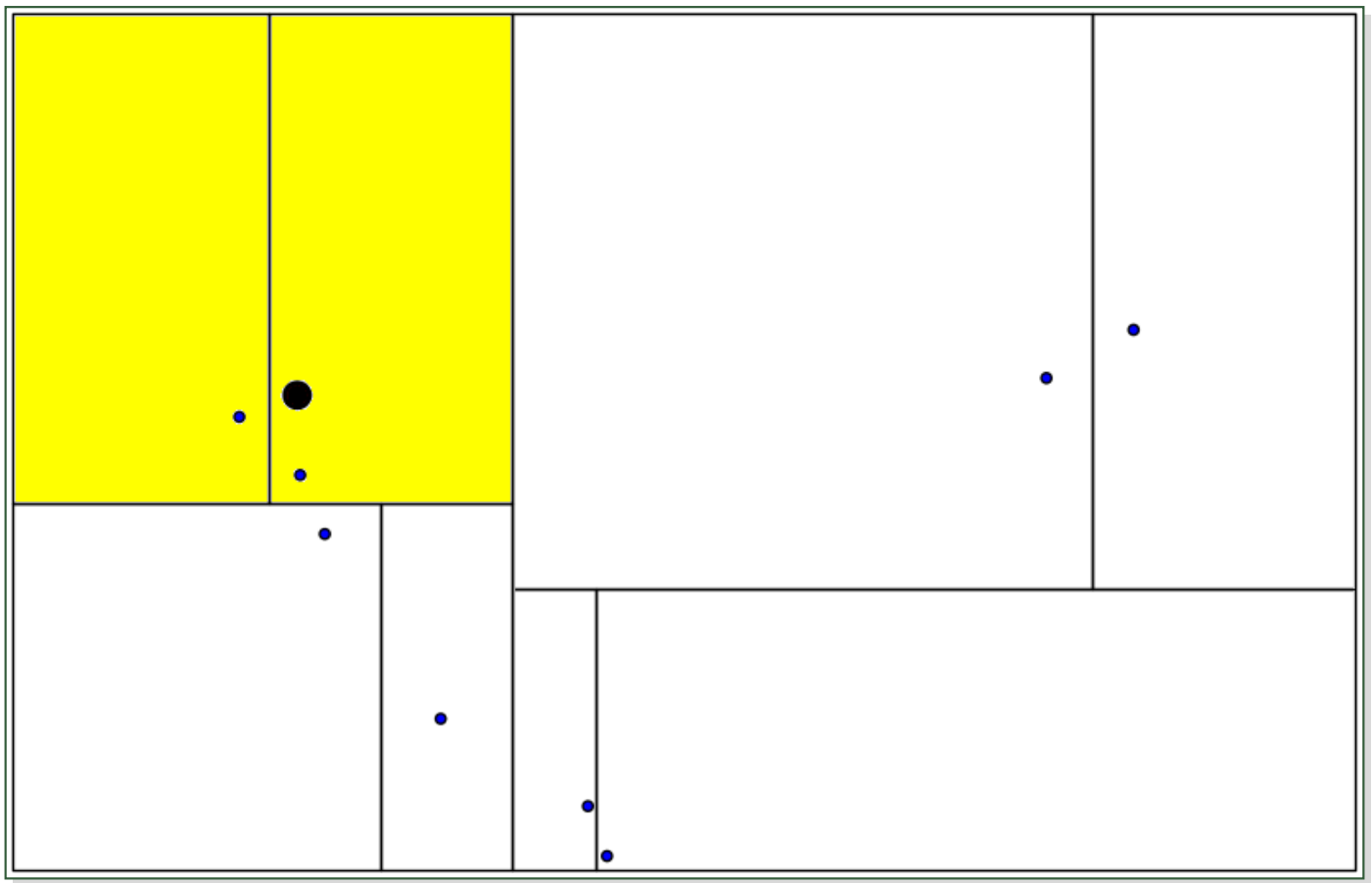




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

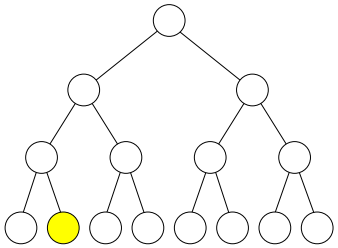
A Classic: *k*-d Trees

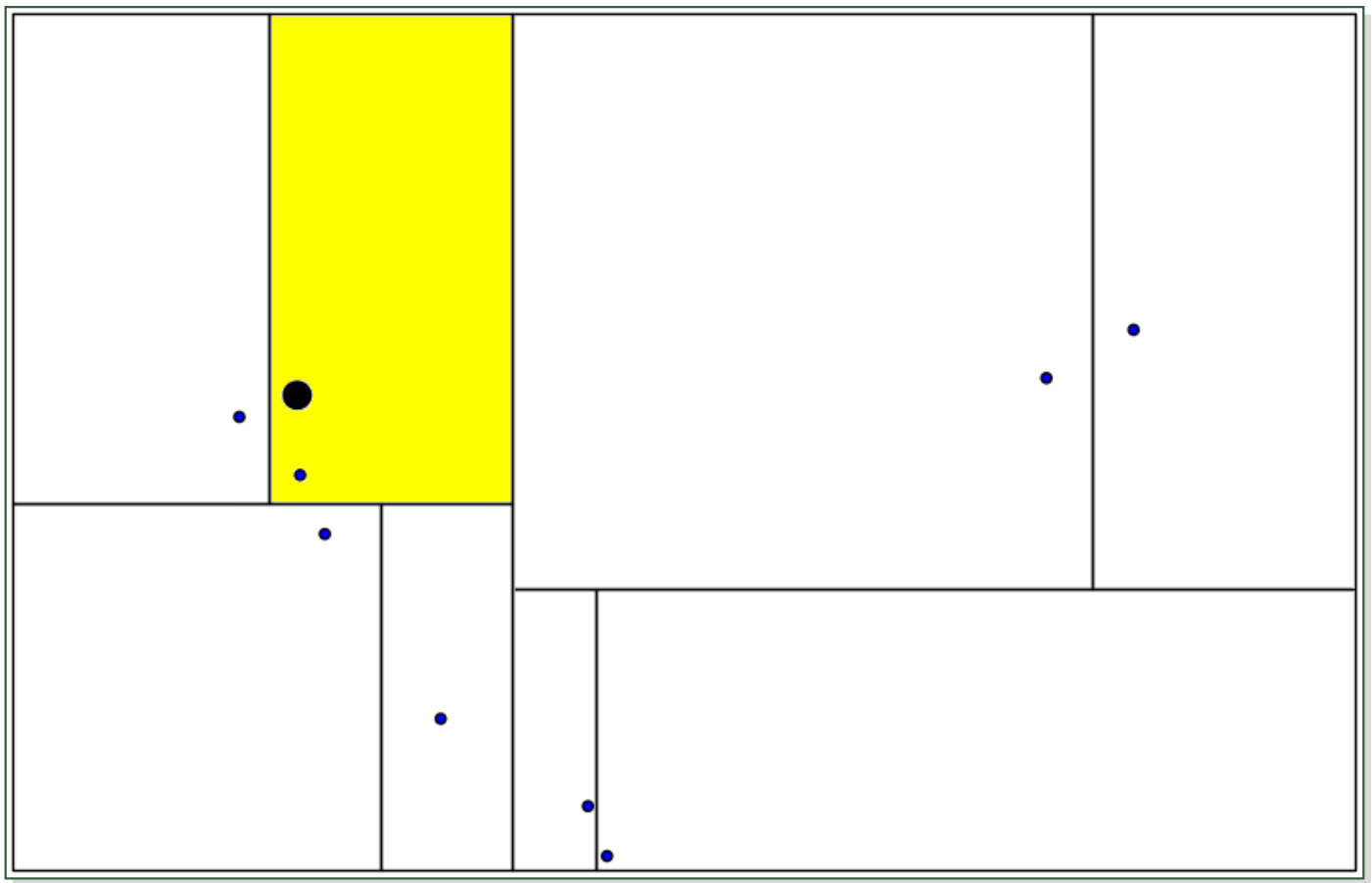




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

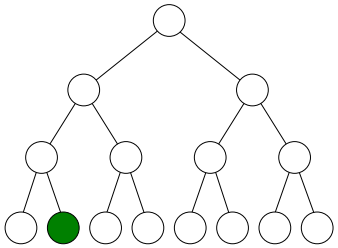
A Classic: *k*-d Trees

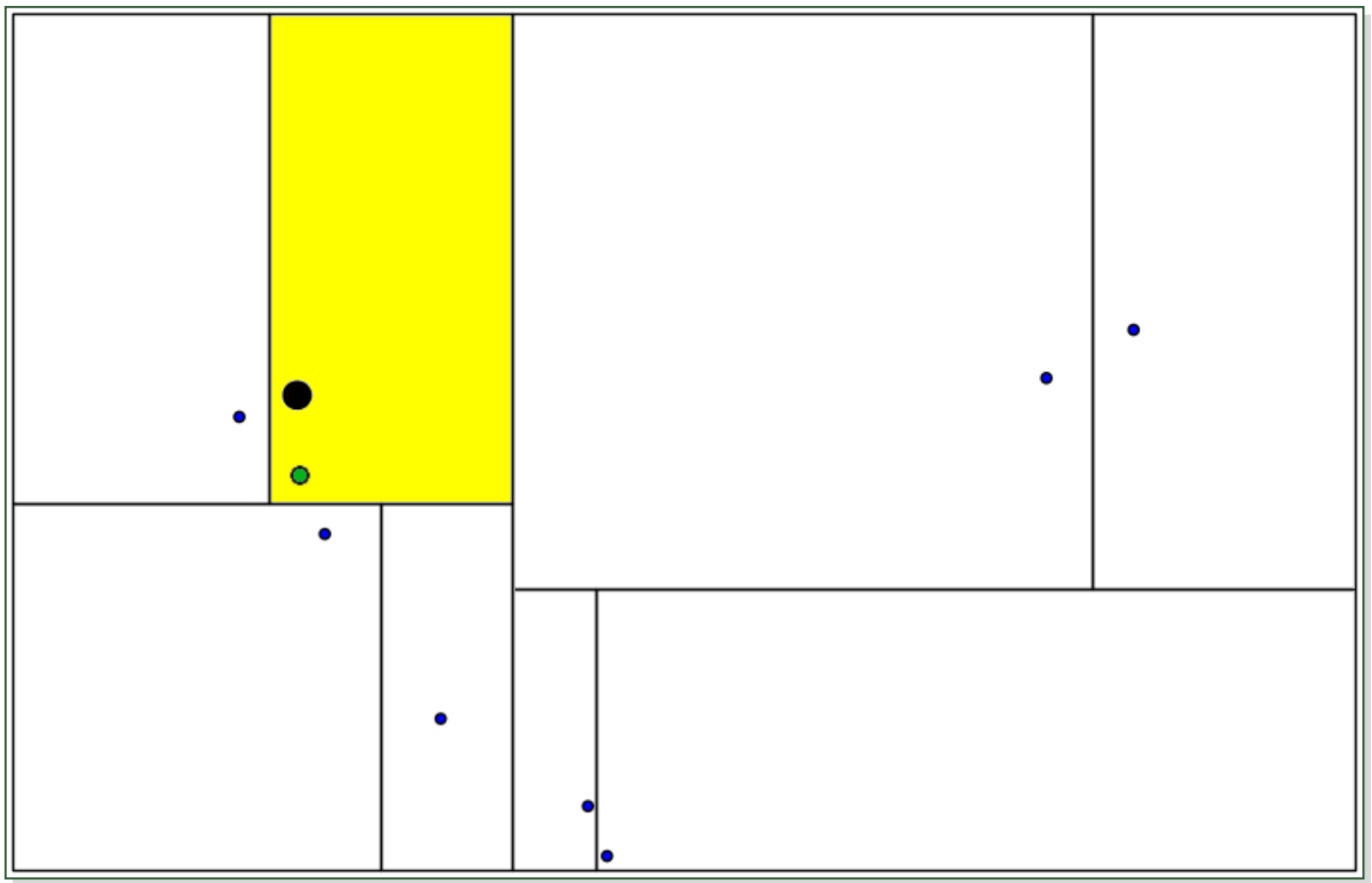




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

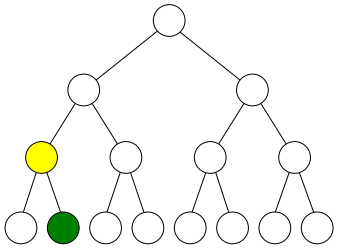
A Classic: *k*-d Trees

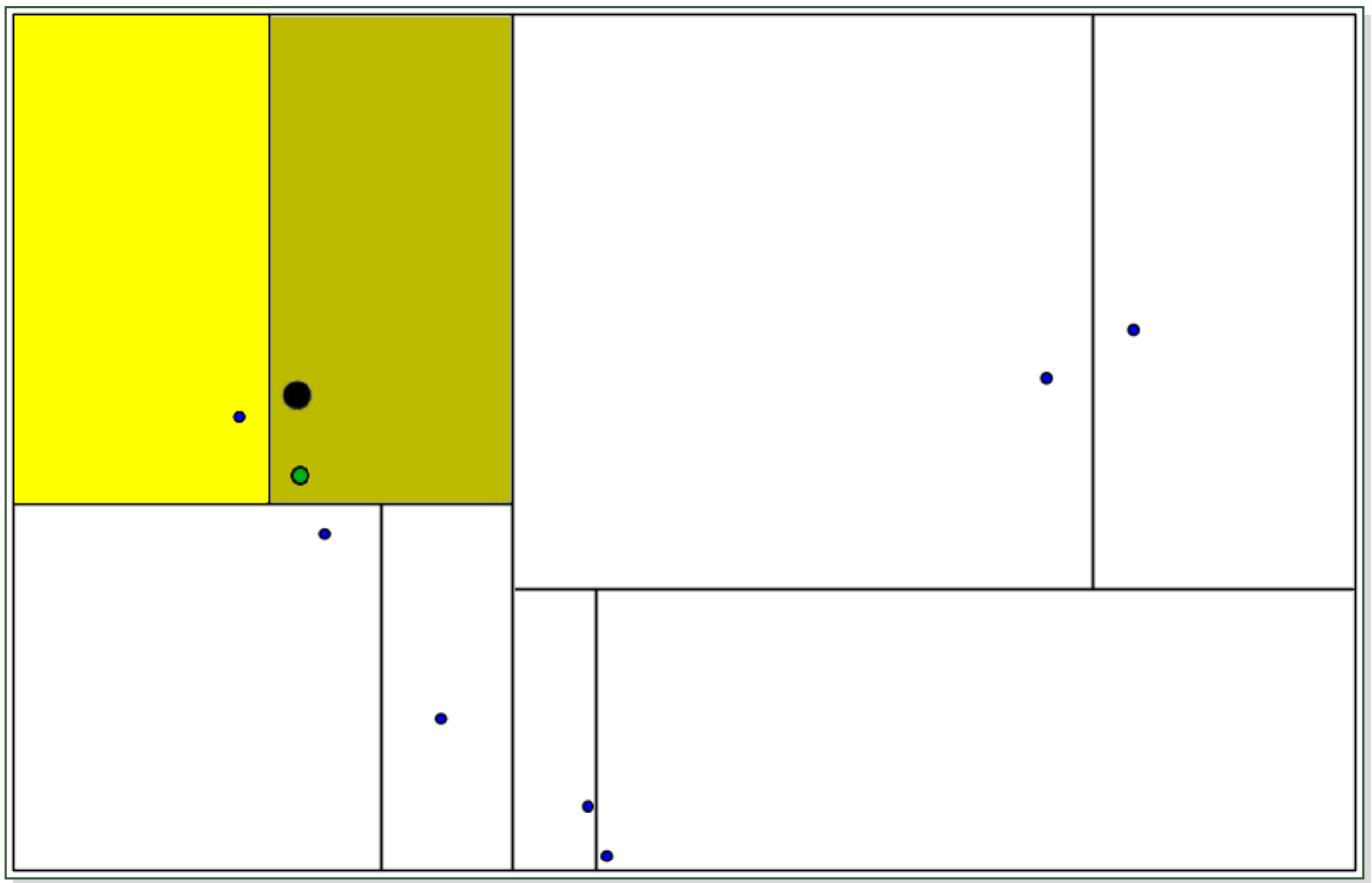




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

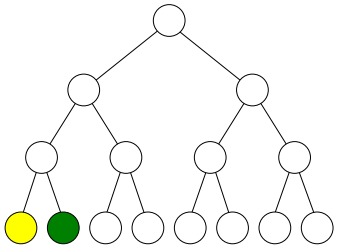
A Classic: *k*-d Trees

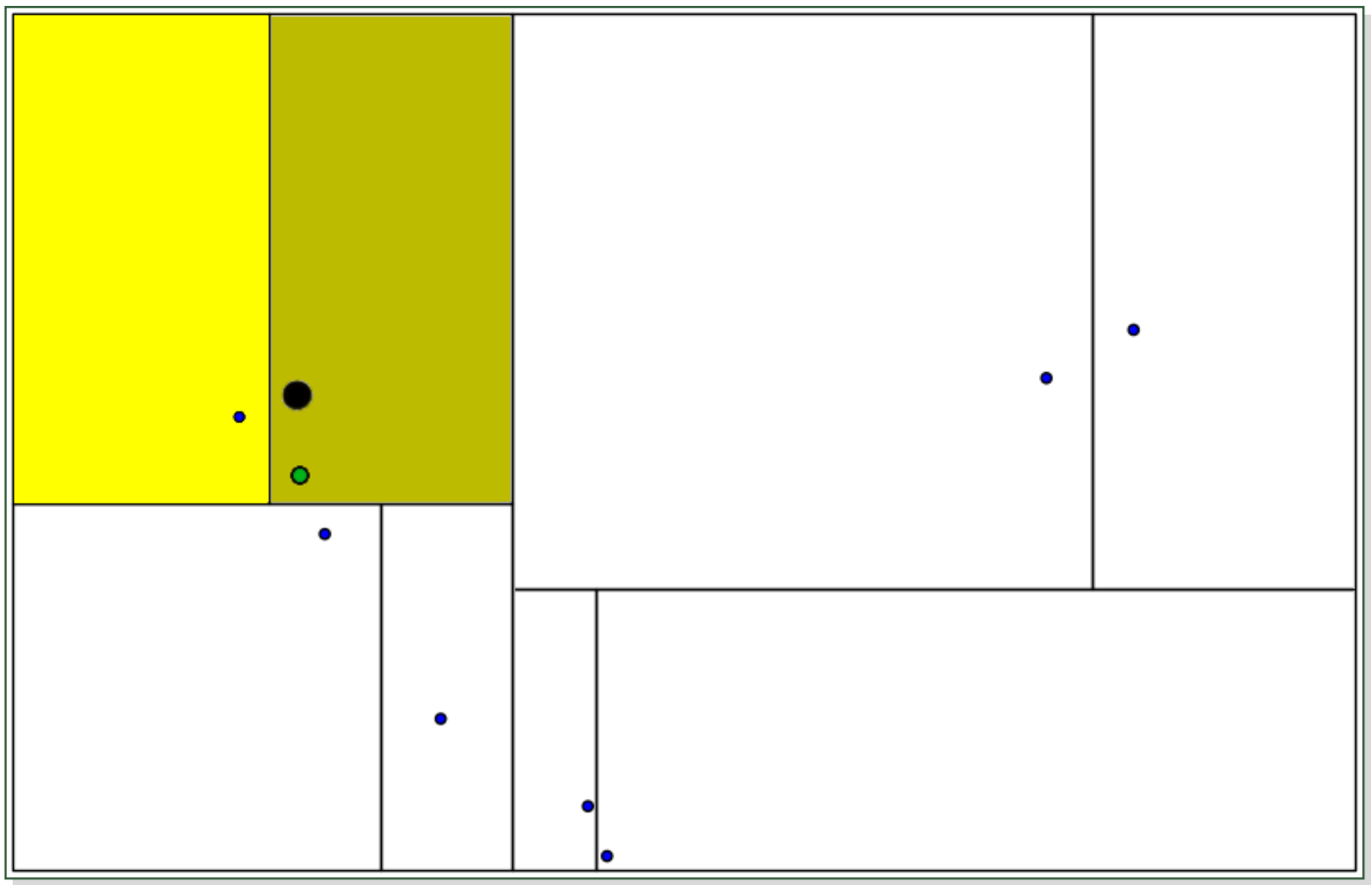




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

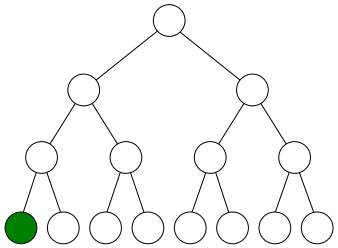
A Classic: *k*-d Trees

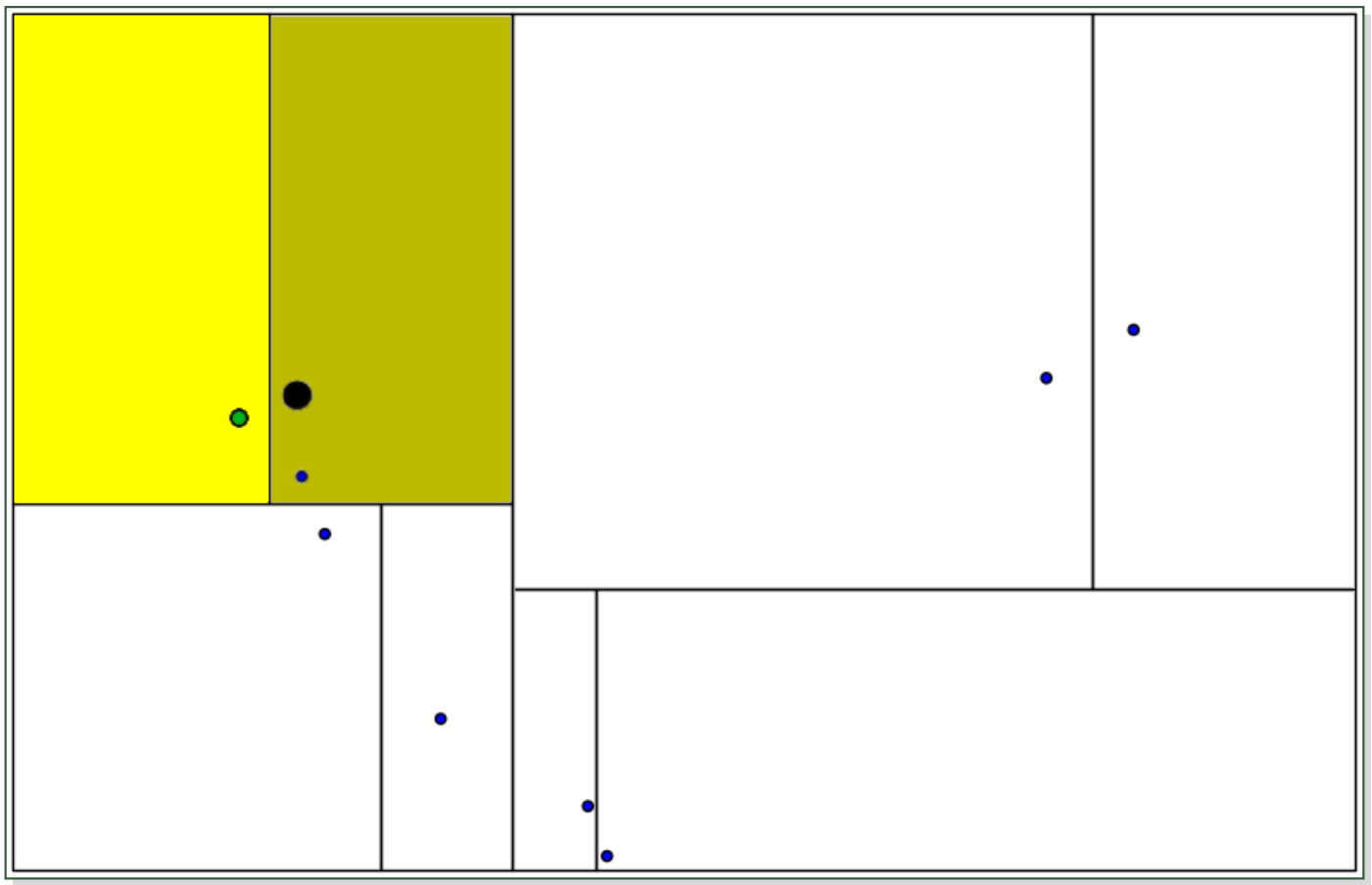




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

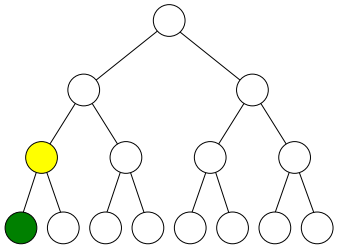
A Classic: *k*-d Trees

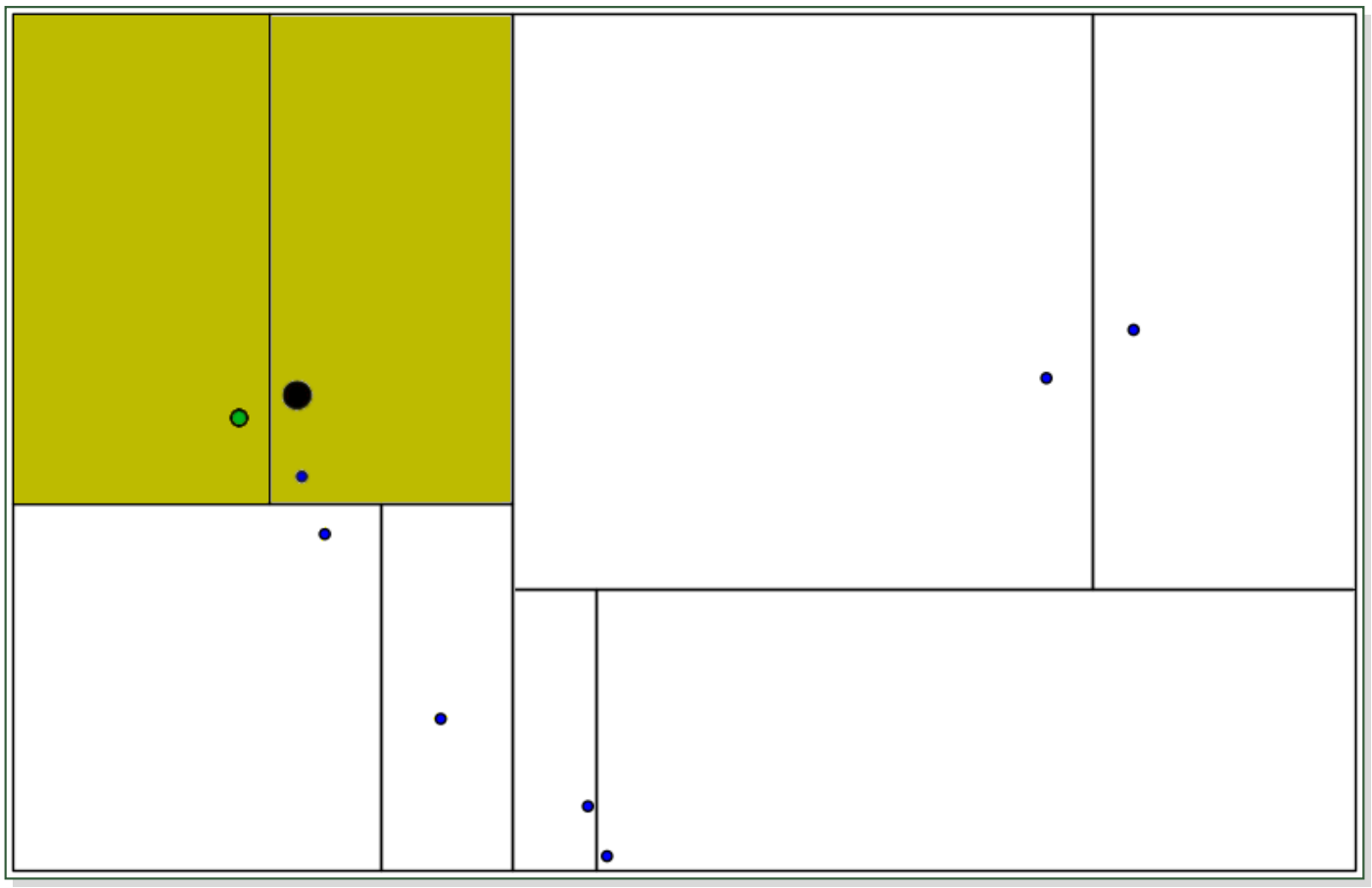




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

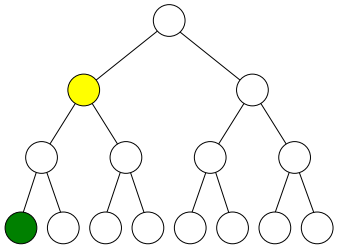
A Classic: *k*-d Trees

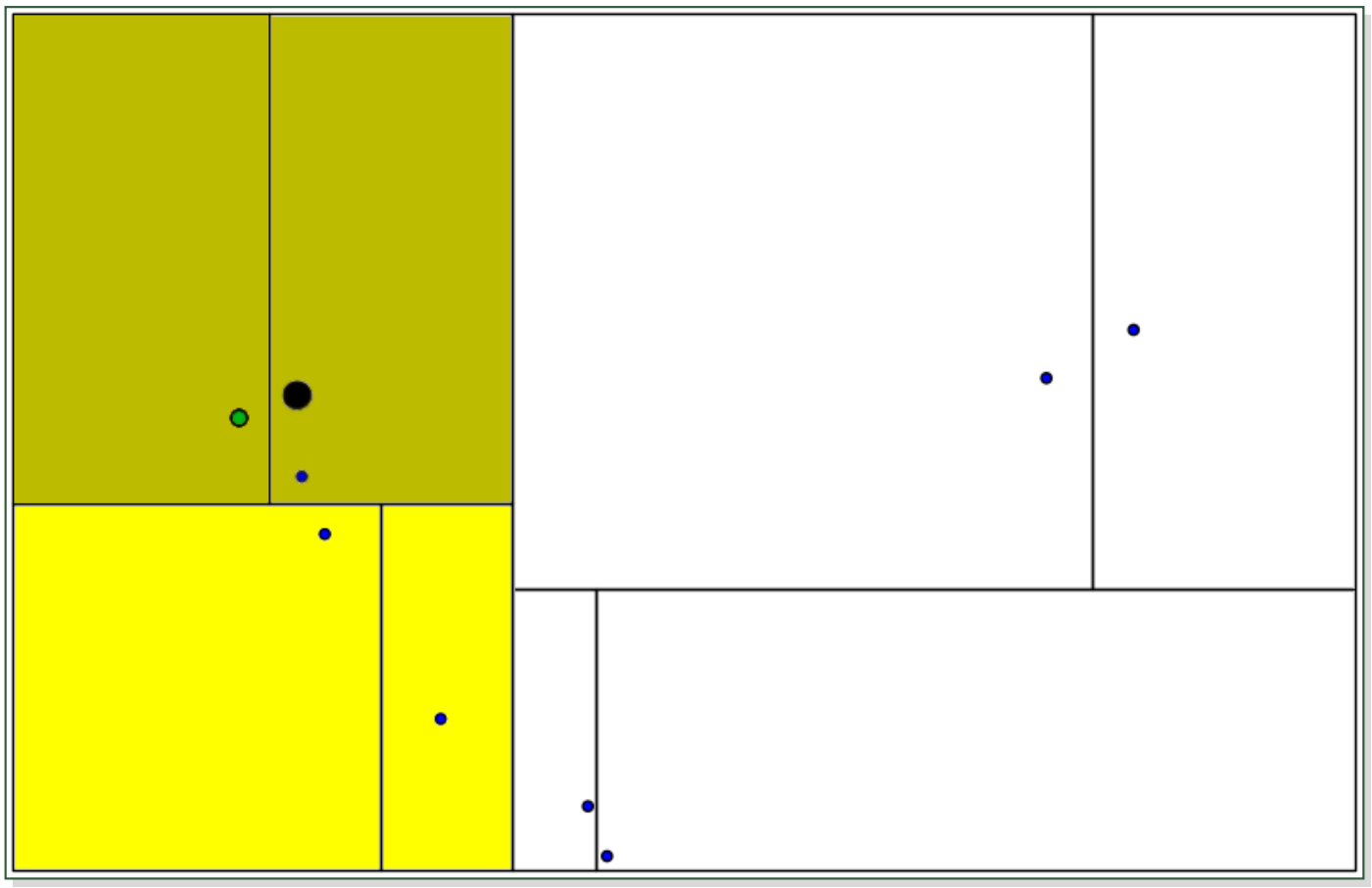




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

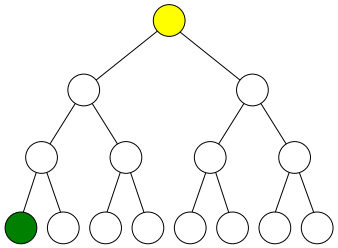
A Classic: *k*-d Trees

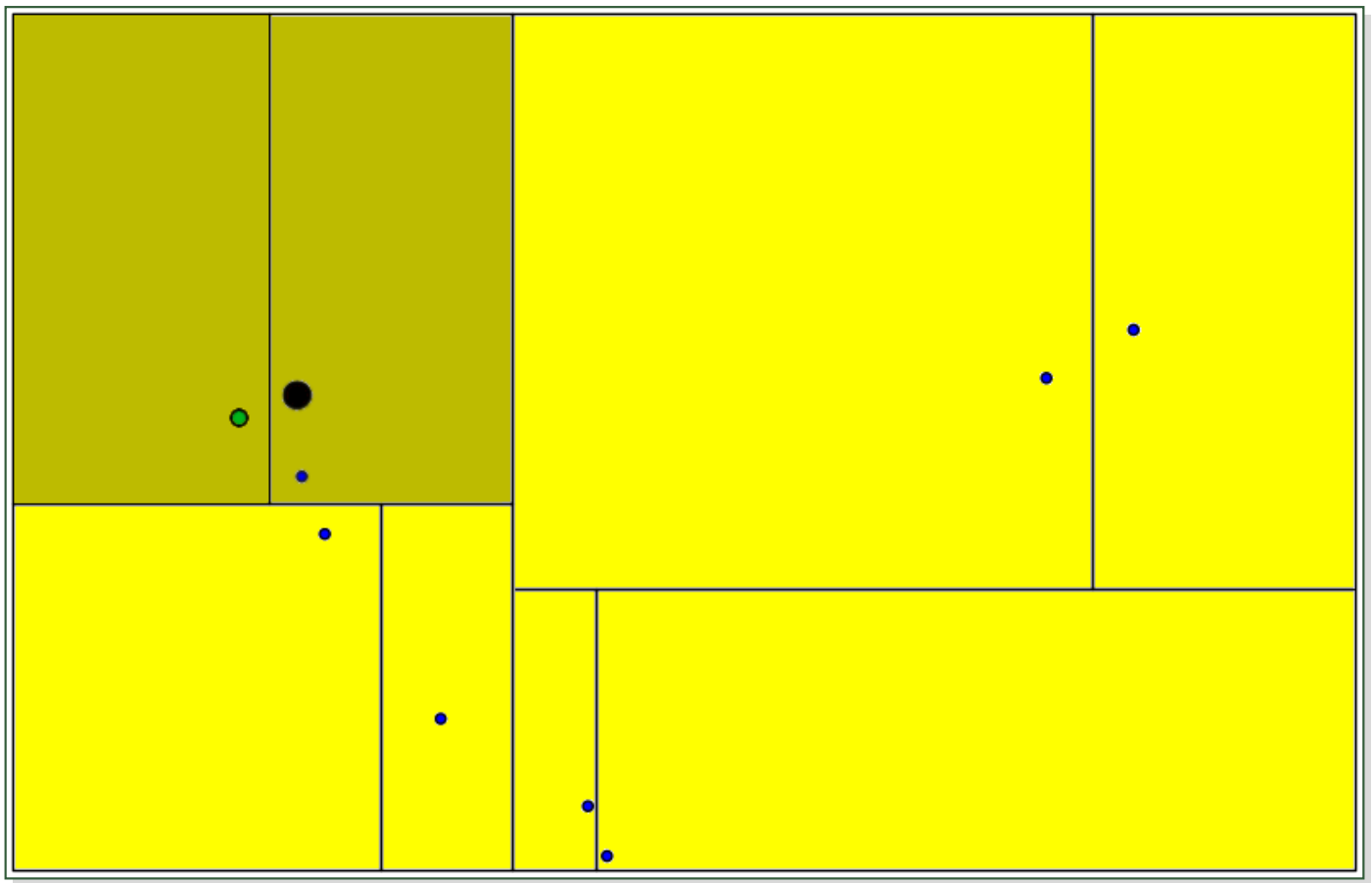




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

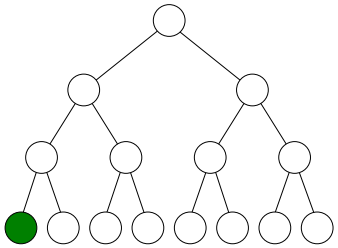
A Classic: *k*-d Trees

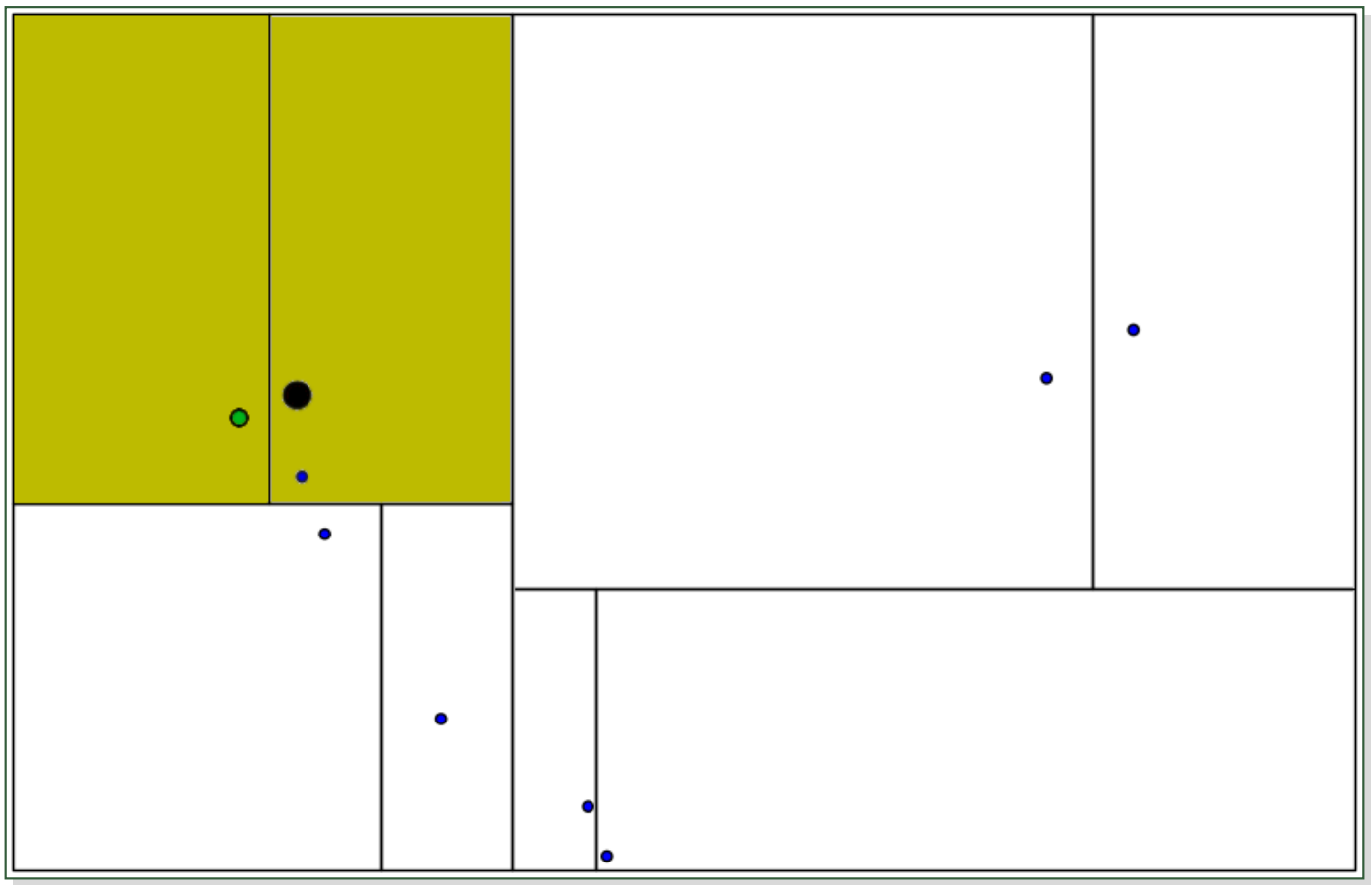




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

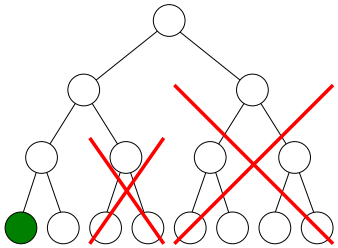
A Classic: *k*-d Trees

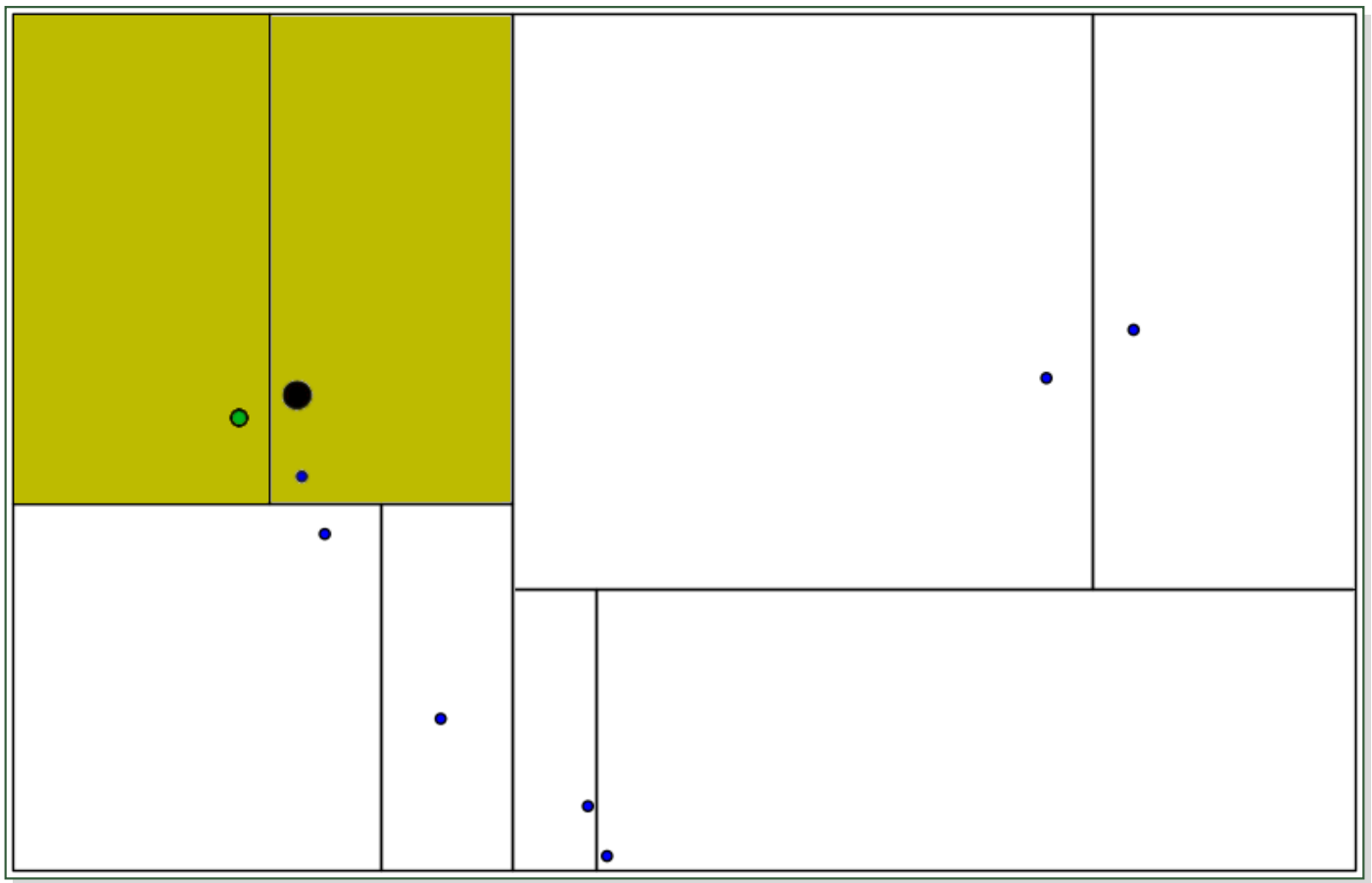




Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

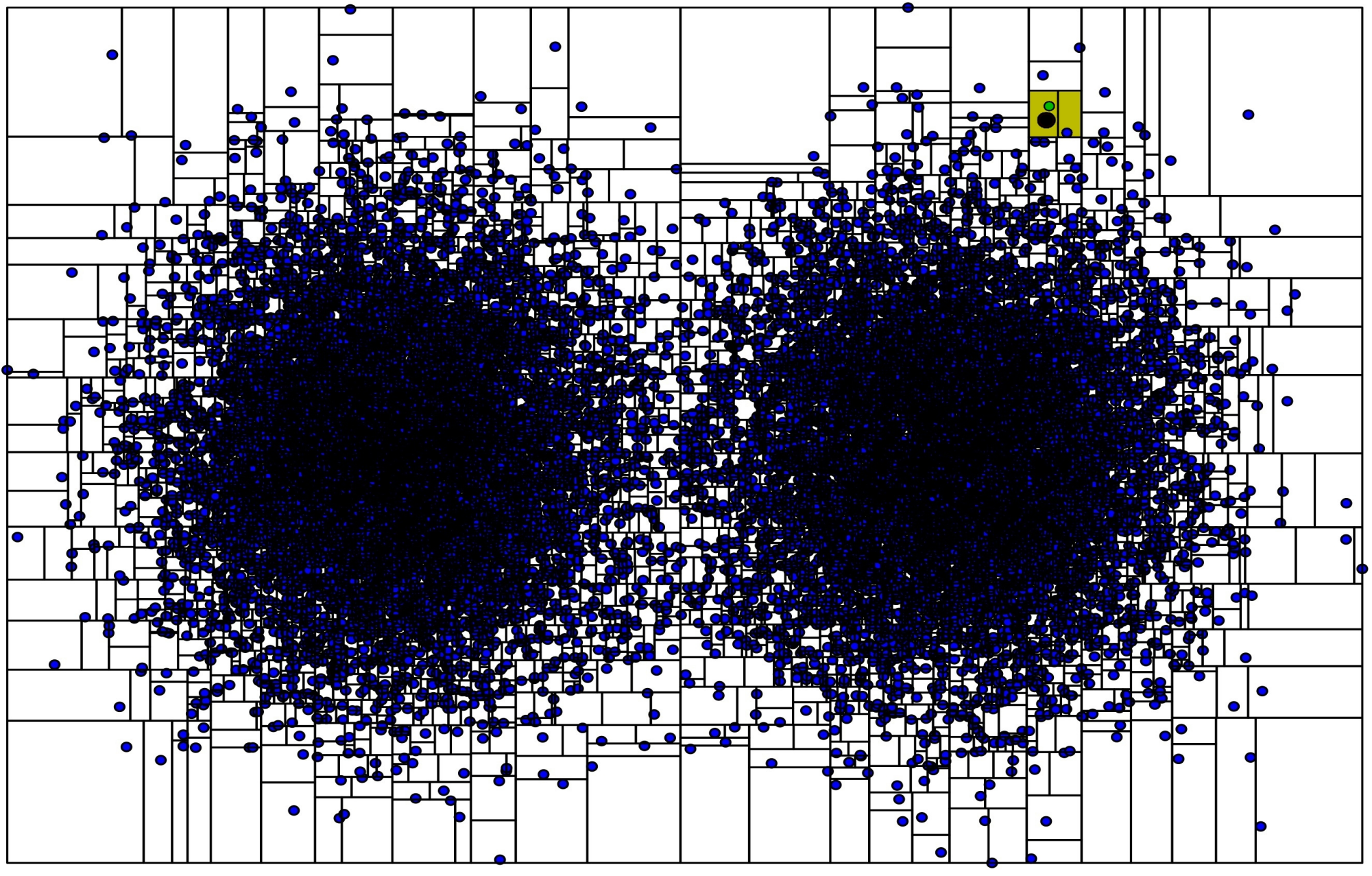
A Classic: *k*-d Trees



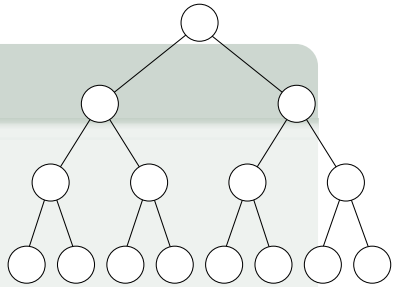


Bentley. *Multidimensional Binary Search Trees Used For Associative Searching*. Communications of the ACM, 1975.

Why They Work in Practice . . .



Construction of k-d Trees



Recursive Tree Construction

Procedure: BUILDKDTREE(S, j)

Require: Point set $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ and current depth j.

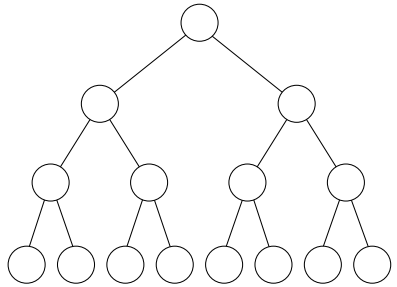
Ensure: K-D tree $\mathcal{T}$ built for S.

1: **if** $|S| = 1$ **then**
2: **return** leaf node storing S (single point)
3: **end if**
4: $ax = j \,\%\, d$
5: m, L, R = FINDMEDIANANDSPLIT(S, ax)
6: $\mathcal{T}_l$ = BUILDKDTREE(L, $j+1$)
7: $\mathcal{T}_r$ = BUILDKDTREE(R, $j+1$)
8: Generate node storing ax, m, and pointers to the trees $\mathcal{T}_l$ and $\mathcal{T}_r$. Let $\mathcal{T}$ denote the resulting tree.
9: **return** $\mathcal{T}$

Let $\mathbf{x}_{j_1}, \ldots, \mathbf{x}_{j_{|S|}}$ be the sequence of all points being sorted according to dimension ax and let m denote the median of this sequence. The procedure FINDMEDIANANDSPLIT returns m as well as the two subsets $L = \{\mathbf{x}_{j_1}, \ldots, \mathbf{x}_{midx}\}$ and $R = \{\mathbf{x}_{midx+1}, \ldots, \mathbf{x}_{|S|}\}$, where $midx = \left\lceil \frac{|S|}{2} \right\rceil$.

Start construction with BUILDKDTREE$(S, 0)$.

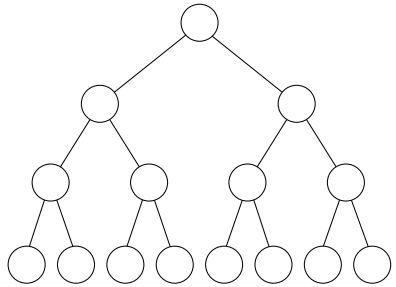
Questions



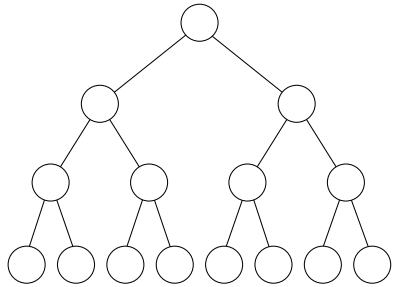
1. How much additional memory is needed for a k-d tree?
2. How much time is needed for the construction phase (worst-case)?
3. What is the worst-case runtime for processing a single query?

Analysis: Nearest Neighbors & k-d Trees

- At each level of the recursion, we spend $O(n)$ time.
  (check: finding median + splitting S can be done in linear time!)

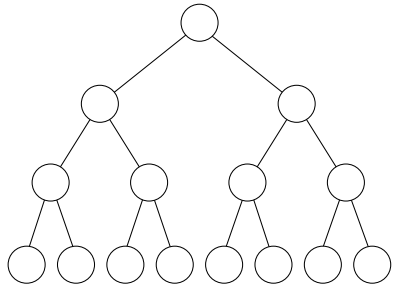


Analysis: Nearest Neighbors & k-d Trees



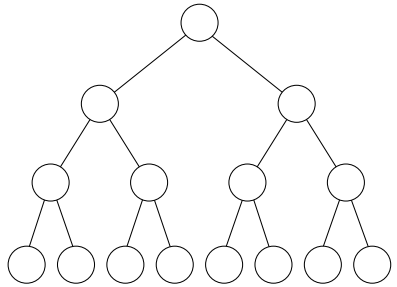
- At each level of the recursion, we spend $O(n)$ time.
  (check: finding median + splitting S can be done in linear time!)
- The recursion ends as soon as a single point is left
  - $\rightarrow$ A k-d tree is complete except for the last level.

Analysis: Nearest Neighbors & k-d Trees



- At each level of the recursion, we spend $O(n)$ time.
  (check: finding median + splitting S can be done in linear time!)
- The recursion ends as soon as a single point is left
  - $\rightarrow$ A k-d tree is complete except for the last level.
  - $\rightarrow$ A k-d tree has height $h = \lceil \log n \rceil$ (starting from $h = 0, 1, 2, \ldots$).

Analysis: Nearest Neighbors & k-d Trees

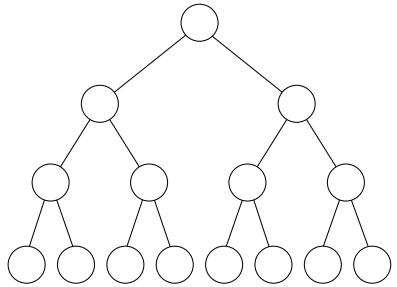


- At each level of the recursion, we spend $O(n)$ time.
  (check: finding median + splitting S can be done in linear time!)
- The recursion ends as soon as a single point is left
  - $\rightarrow$ A k-d tree is complete except for the last level.
  - $\rightarrow$ A k-d tree has height $h = \lceil \log n \rceil$ (starting from $h = 0, 1, 2, \ldots$).
- The runtime for the construction is given by (c constant, large enough):

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \\ &= 2 \cdot 2 \cdot T\left(\frac{n}{4}\right) + 2 \cdot c \cdot \frac{n}{2} + c \cdot n \\ &= \ldots \\ &= \sum_{i=0}^{\lceil \log n \rceil - 1} 2^i \cdot c \cdot \frac{n}{2^i} \end{aligned}$$

Thus, $T(n) \in O(n \log n)$ time for n points.

Analysis: Nearest Neighbors & k-d Trees

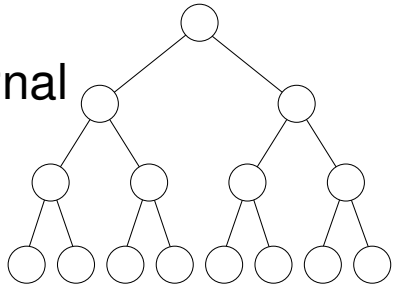


- At each level of the recursion, we spend $O(n)$ time.
  (check: finding median + splitting S can be done in linear time!)
- The recursion ends as soon as a single point is left
  - $\rightarrow$ A k-d tree is complete except for the last level.
  - $\rightarrow$ A k-d tree has height $h = \lceil \log n \rceil$ (starting from $h = 0, 1, 2, \ldots$).
- The runtime for the construction is given by (c constant, large enough):

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \\ &= 2 \cdot 2 \cdot T\left(\frac{n}{4}\right) + 2 \cdot c \cdot \frac{n}{2} + c \cdot n \\ &= \ldots \\ &= \sum_{i=0}^{\lceil \log n \rceil - 1} 2^i \cdot c \cdot \frac{n}{2^i} \end{aligned}$$

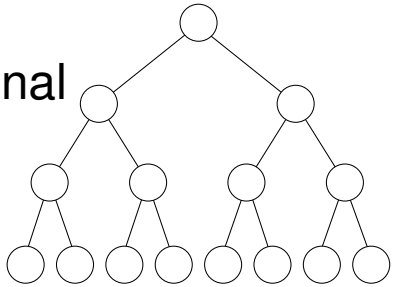
  Thus, $T(n) \in O(n \log n)$ time for n points.
- In practice, one often stops the recursion as soon as a pre-defined number of points is left (e.g., $n = 50$). During the search, one has to compare with all the points that are stored in a leaf (brute-force).

Analysis: Nearest Neighbors & k-d Trees



- A k-d tree has at most $2^{\lceil \log n \rceil}$ leaves and $2^{\lceil \log n \rceil} - 1$ internal nodes. Hence: $O(n)$ additional space.
- Nearest neighbor search takes $O(\log n)$ time in practice per query point (for low-dimensional spaces up to, e.g., $\mathbb{R}^{20}$). Worst-case runtime ist still $O(n)$ per query. (not better than brute-force!)

Analysis: Nearest Neighbors & k-d Trees



- A k-d tree has at most $2^{\lceil \log n \rceil}$ leaves and $2^{\lceil \log n \rceil} - 1$ internal nodes. Hence: $O(n)$ additional space.
- Nearest neighbor search takes $O(\log n)$ time in practice per query point (for low-dimensional spaces up to, e.g., $\mathbb{R}^{20}$). Worst-case runtime ist still $O(n)$ per query. (not better than brute-force!)

Summary: k-d Trees

Let $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ be a set of n points. A k-d tree for X can be constructed in $O(n \log n)$ time and needs $O(n)$ additional space. Nearest neighbor search takes $O(\log n)$ time in practice per query. Worst-case runtime per query is $O(n)$.

Metric Trees

Recursive Tree Construction

Procedure: BUILDMETRICTREE(S)

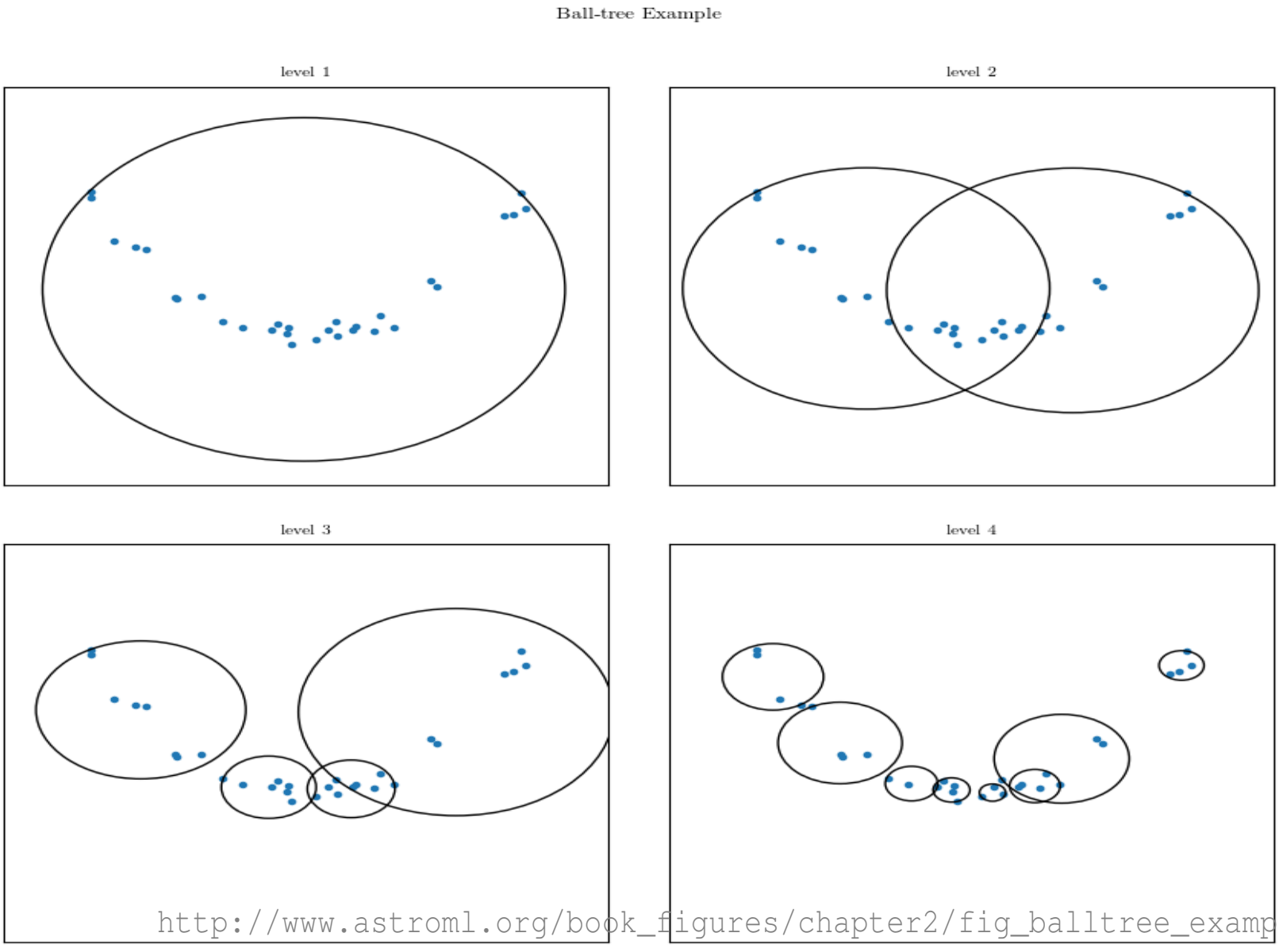
Require: Point set $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$

Ensure: Metric tree $\mathcal{T}$ for S.

1: **if** $|S| \leq M$ **then**
2: **return** leaf node storing the $|S|$ points
3: **end if**
4: Let *ax* be the dimension of greatest spread
5: m, L, R = FINDMEDIANANDSPLIT(S, ax)
6: $\mathcal{T}_l$ = BUILDMETRICTREE(L)
7: $\mathcal{T}_r$ = BUILDMETRICTREE(R)
8: Generate node storing the center of the ball (mean of S), the radius r, and pointers to the trees $\mathcal{T}_l$ and $\mathcal{T}_r$. Let $\mathcal{T}$ denote the resulting tree.
9: **return** $\mathcal{T}$

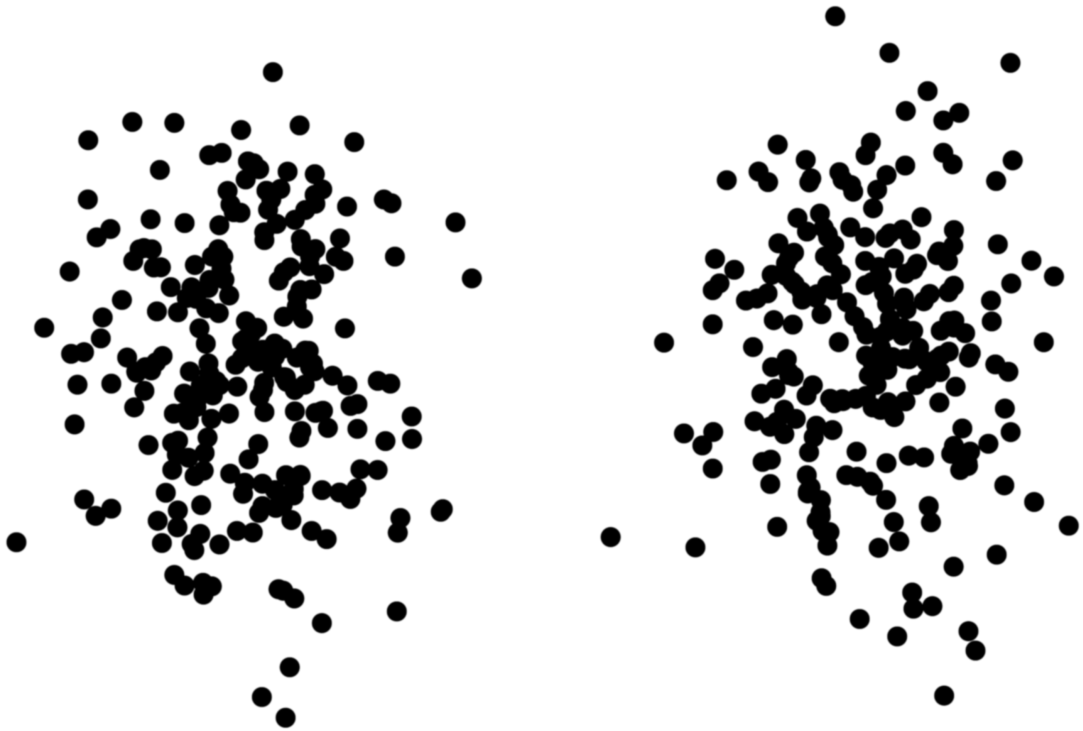
Omohundrol. *Five Balltree Construction Algorithms*. Technical report, 1989.

Metric Trees



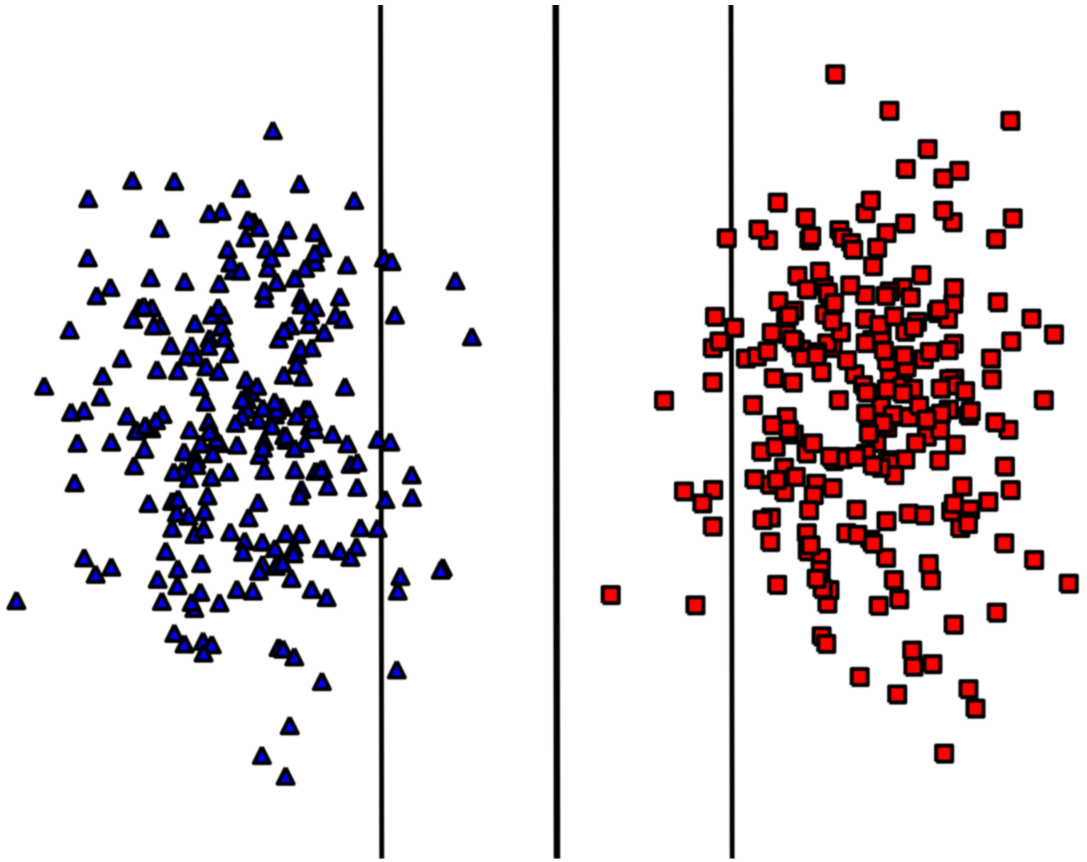
http://www.astroml.org/book_figures/chapter2/fig_balltree_example.html

Maximum Margin Clustering



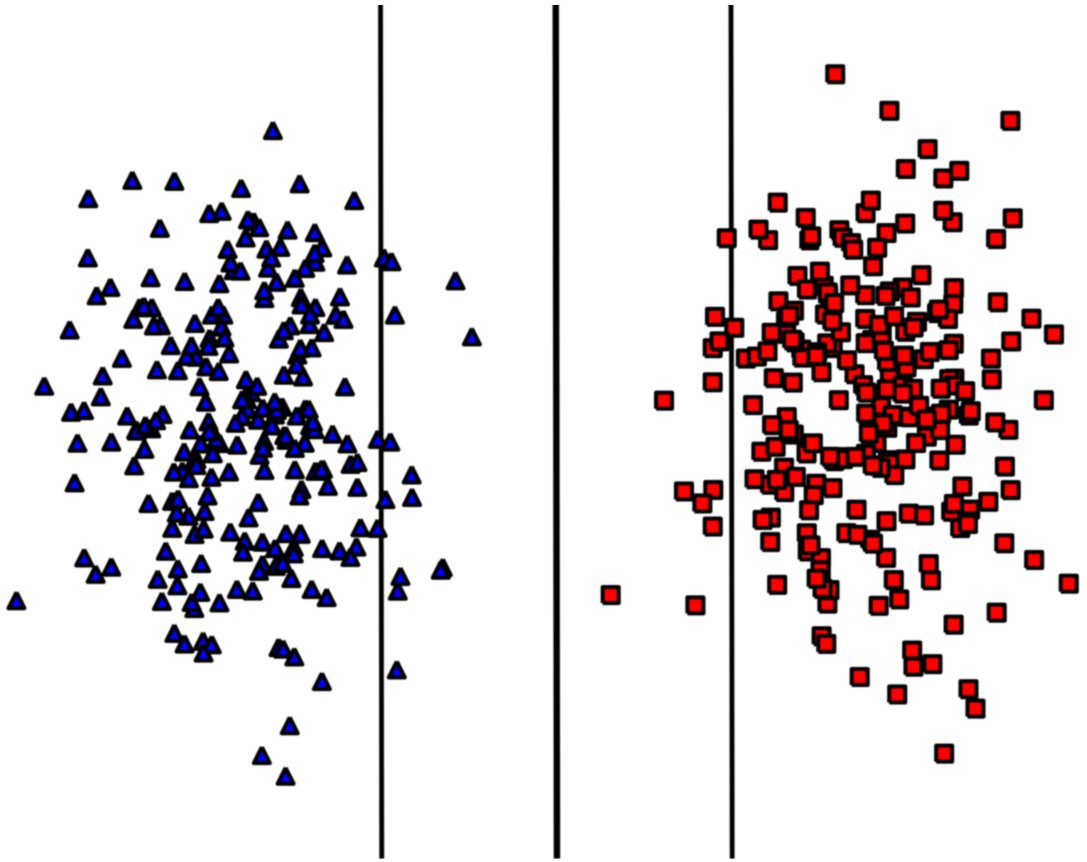
Partition the points into two classes such that a subsequent application of a support vector machine yields the best overall result!

Maximum Margin Clustering



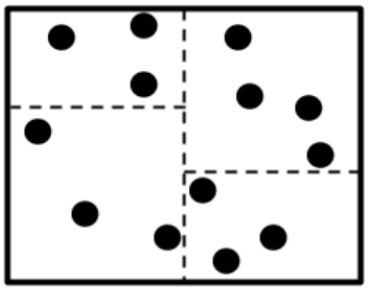
Partition the points into two classes such that a subsequent application of a support vector machine yields the best overall result!

Maximum Margin Clustering

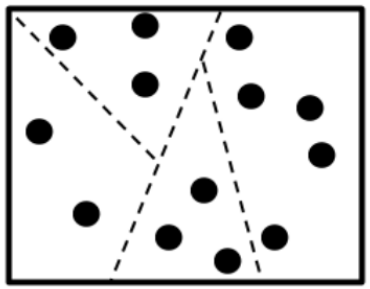


Brute-Force: Test all possible 2^n partitions via a support vector machine!

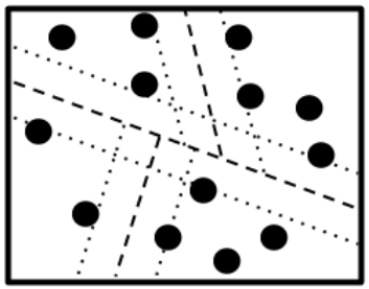
Max-Margin Trees



(a) *kd*-tree



(b) *RP*-tree



(c) *MM*-tree

Figure 1: **Binary space-partitioning trees.**

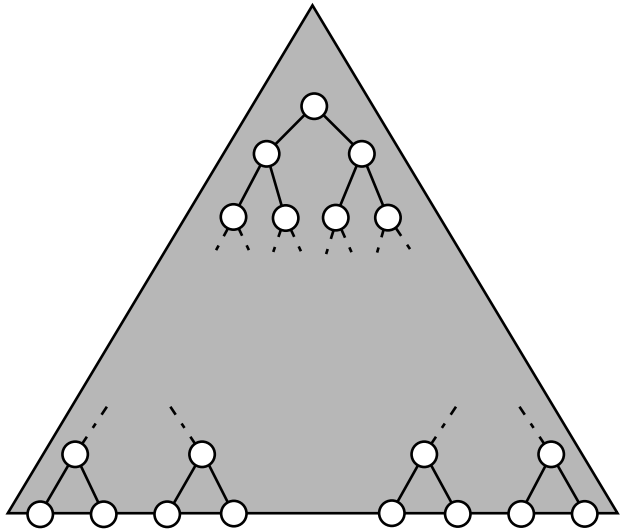
Maximum Margin Clustering

$$\begin{aligned} \underset{\substack{\mathbf{y}\in\{-1,+1\}^n,\\ \mathbf{w}\in\mathcal{H},\, b\in\mathbb{R},\, \xi\in\mathbb{R}^n}}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i\big(\langle \mathbf{w}, \Phi(\mathbf{x}_i)\rangle\big) + b\big) \geq 1-\xi_i,\ \xi_i \geq 0 \\ \text{and} \quad & -1 \leq \frac{1}{n}\sum_{i=1}^{n} y_i \leq 1 \end{aligned}$$

Ram and Gray. *Which Space Partitioning Tree to Use for Search?* NIPS, 2013.

Outline

Parallel Search and *k*-d Trees?

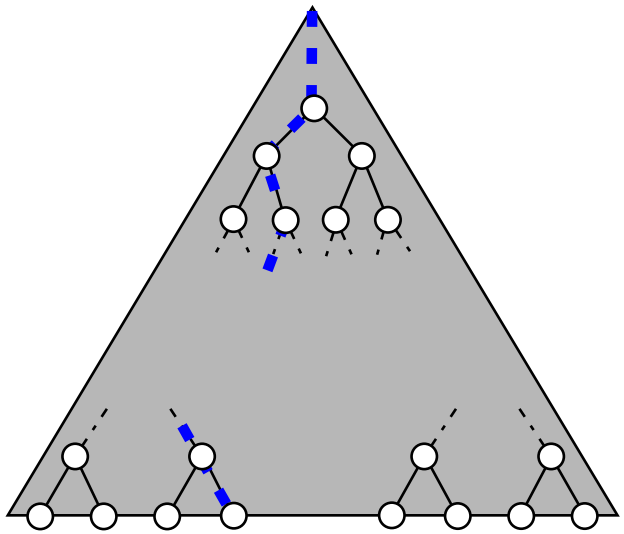


Multiple Instruction, Multiple Data (MIMD)

A typical parallel *k*-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously.

Thread=Program that is executed by a single processor core . . .

Parallel Search and *k*-d Trees?

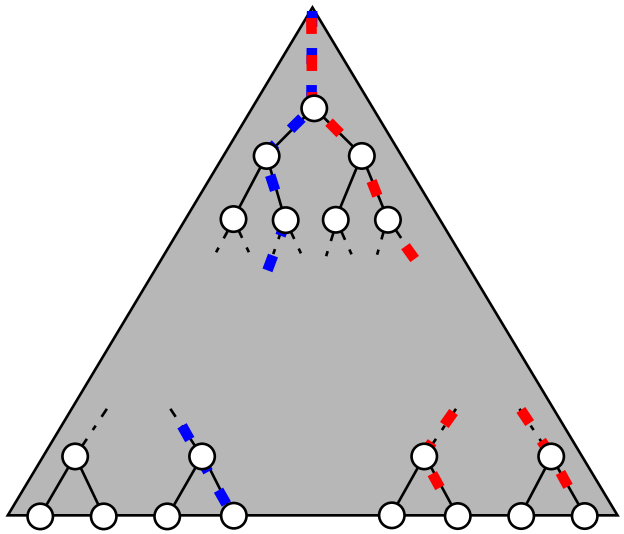


Multiple Instruction, Multiple Data (MIMD)

A typical parallel *k*-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously.

Thread=Program that is executed by a single processor core . . .

Parallel Search and *k*-d Trees?

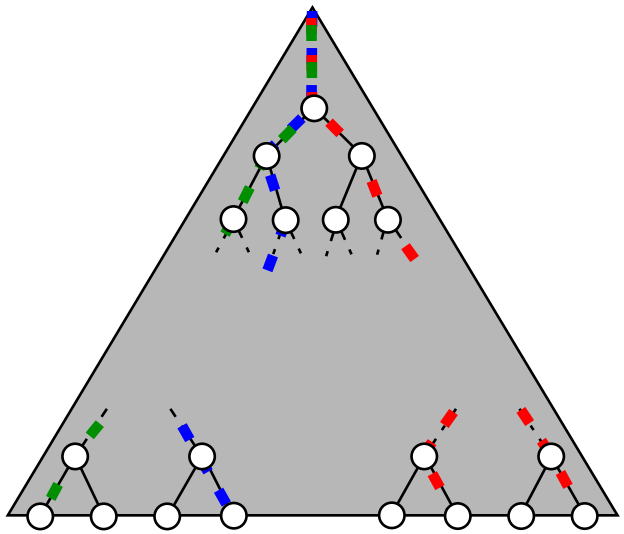


Multiple Instruction, Multiple Data (MIMD)

A typical parallel *k*-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously.

Thread=Program that is executed by a single processor core . . .

Parallel Search and *k*-d Trees?

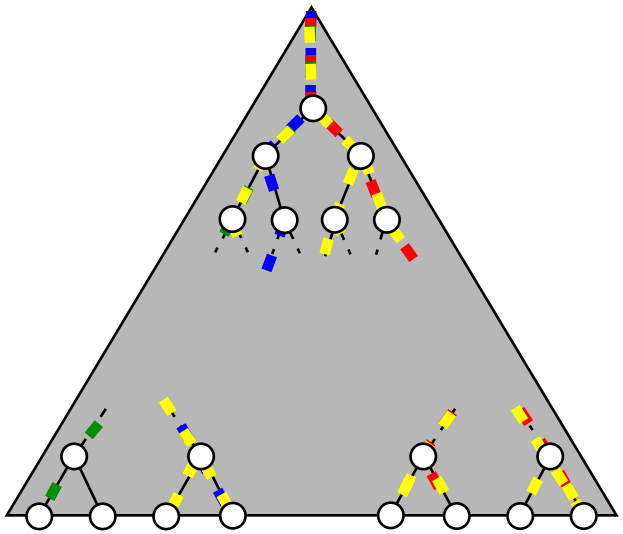


Multiple Instruction, Multiple Data (MIMD)

A typical parallel *k*-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously.

Thread=Program that is executed by a single processor core . . .

Parallel Search and *k*-d Trees?



Multiple Instruction, Multiple Data (MIMD)

A typical parallel *k*-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously.

Thread=Program that is executed by a single processor core . . .

Parallel Search and *k*-d Trees

Example

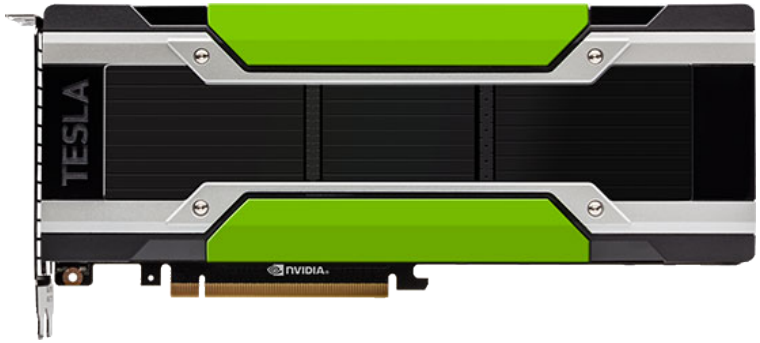
```
1  # https://github.com/gieseke/bufferkdtree/tree/master/examples
2  import time
3  from bufferkdtree import NearestNeighbors
4  import generate
5
6  Xtrain, Ytrain, Xtest = generate.get_data_set(
7                data_set="psf_model_mag",
8                NUM_TRAIN=1000000,
9                NUM_TEST=1000000)
10
11 print("Using n_jobs={}".format(n_jobs))
12 nbrs = NearestNeighbors(n_neighbors=10,
13                algorithm="kd_tree",
14                leaf_size=32,
15                n_jobs=1)
16 nbrs.fit(Xtrain)
17
18 start_time = time.time()
19 _, _ = nbrs.kneighbors(Xtest)
20 end_time = time.time()
21 print("Testing time: {}".format((end_time - start_time)))
```

Parallel Search and *k*-d Trees

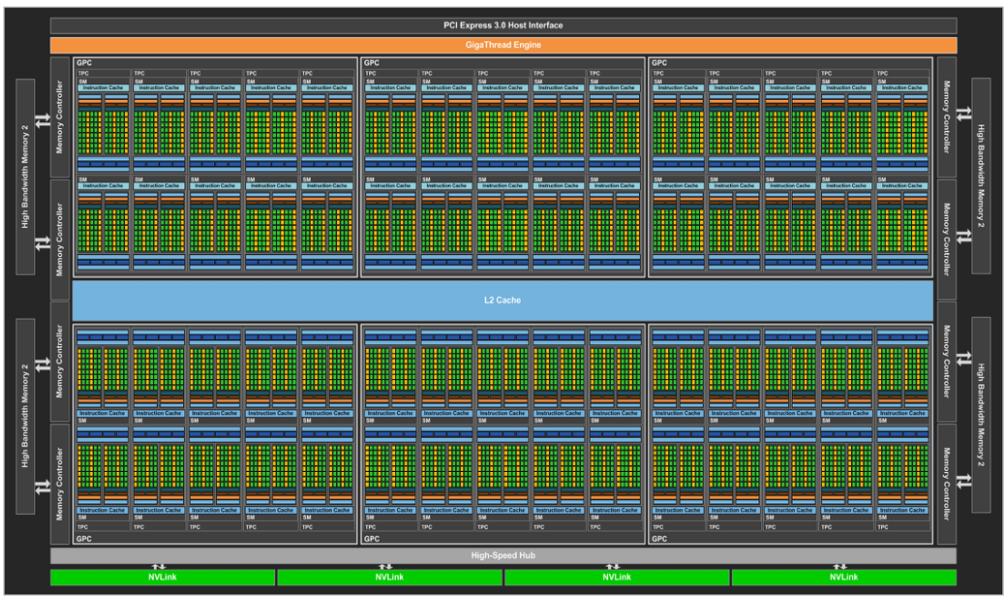
```
(buffer_env) cooluser@gpuserver:~/parallel$ python kdtrees.py
/home/cooluser/parallel/generate.py:77: RuntimeWarning: invalid value encountered in greater
  selector = numpy.min(X,axis=1) > -5000
Using n_jobs=1
Testing time: 130.406774
(buffer_env) cooluser@gpuserver:~/parallel$ python kdtrees.py
/home/cooluser/parallel/generate.py:77: RuntimeWarning: invalid value encountered in greater
  selector = numpy.min(X,axis=1) > -5000
Using n_jobs=6
Testing time: 22.068576
(buffer_env) cooluser@gpuserver:~/parallel$
```

6 Cores, speed-up: ≈ 6

Recap: GPUs (2018)

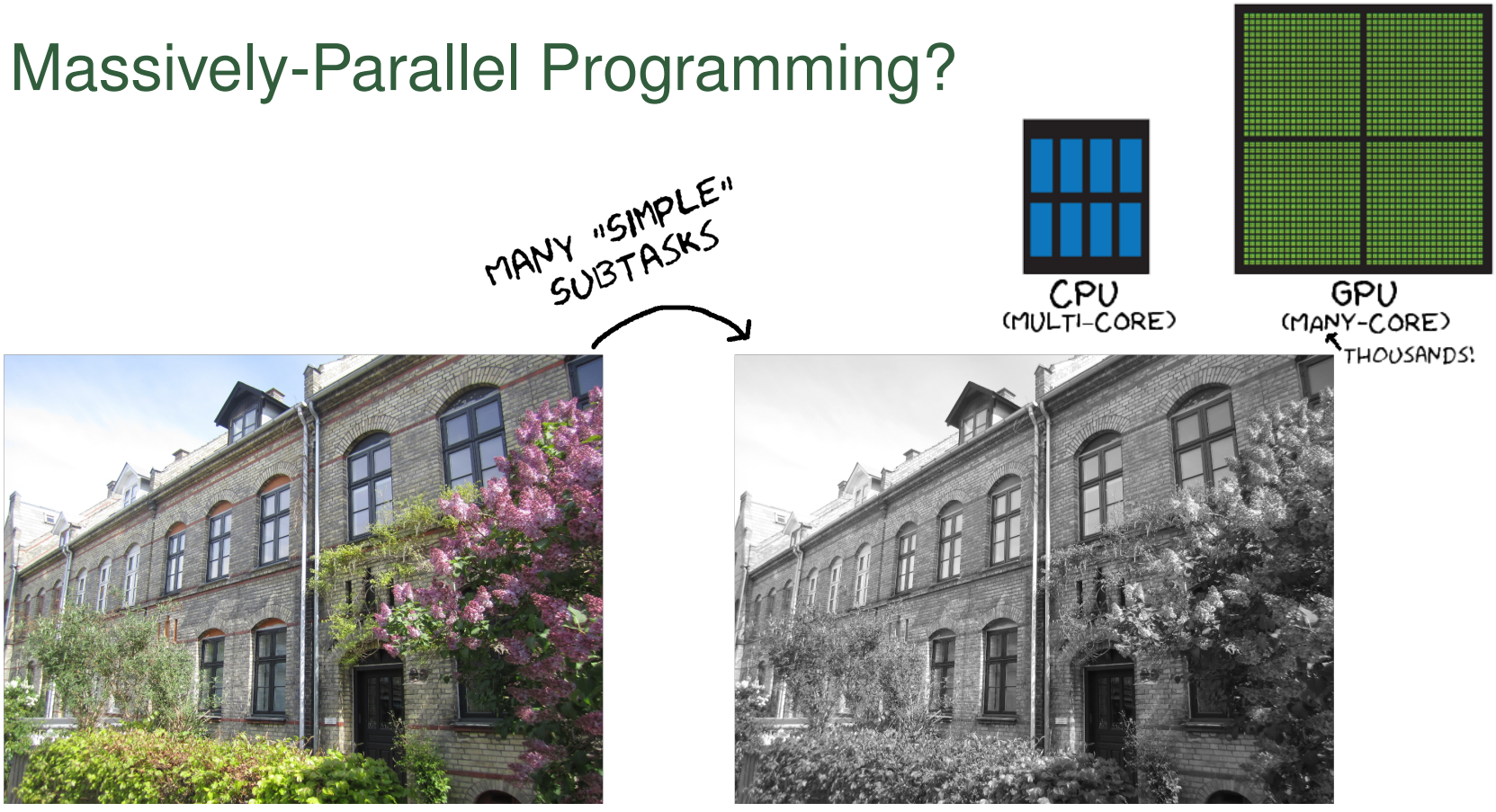


- Thousands of cores
  (e.g., P100: 3,584)
- Clock speed: $\approx$ 0.9 to 1.5GHz
- Implementing efficient GPU code is more difficult
  (e.g., cores execute instructions simultaneously, data access, . . .)
- Power consumption: 20-350W (e.g., 250W for P100)
- Fast caches L1, L2
  (e.g., L2 $\approx$ 4MB)
- Fast onboard memory
  (e.g., 16GB with 732GB/s)
- Host $\leftrightarrow$ GPU memory transfer
  (e.g., 12GB/s)
- GFLOPs: 5000+



https://devblogs.nvidia.com/parallelforall/inside-pascal/
https://www.microway.com/hpc-tech-tips/nvidia-tesla-p100-nvlink-16gb-gpu-accelerator-pascal-gp100-sxm2-close/

Massively-Parallel Programming?

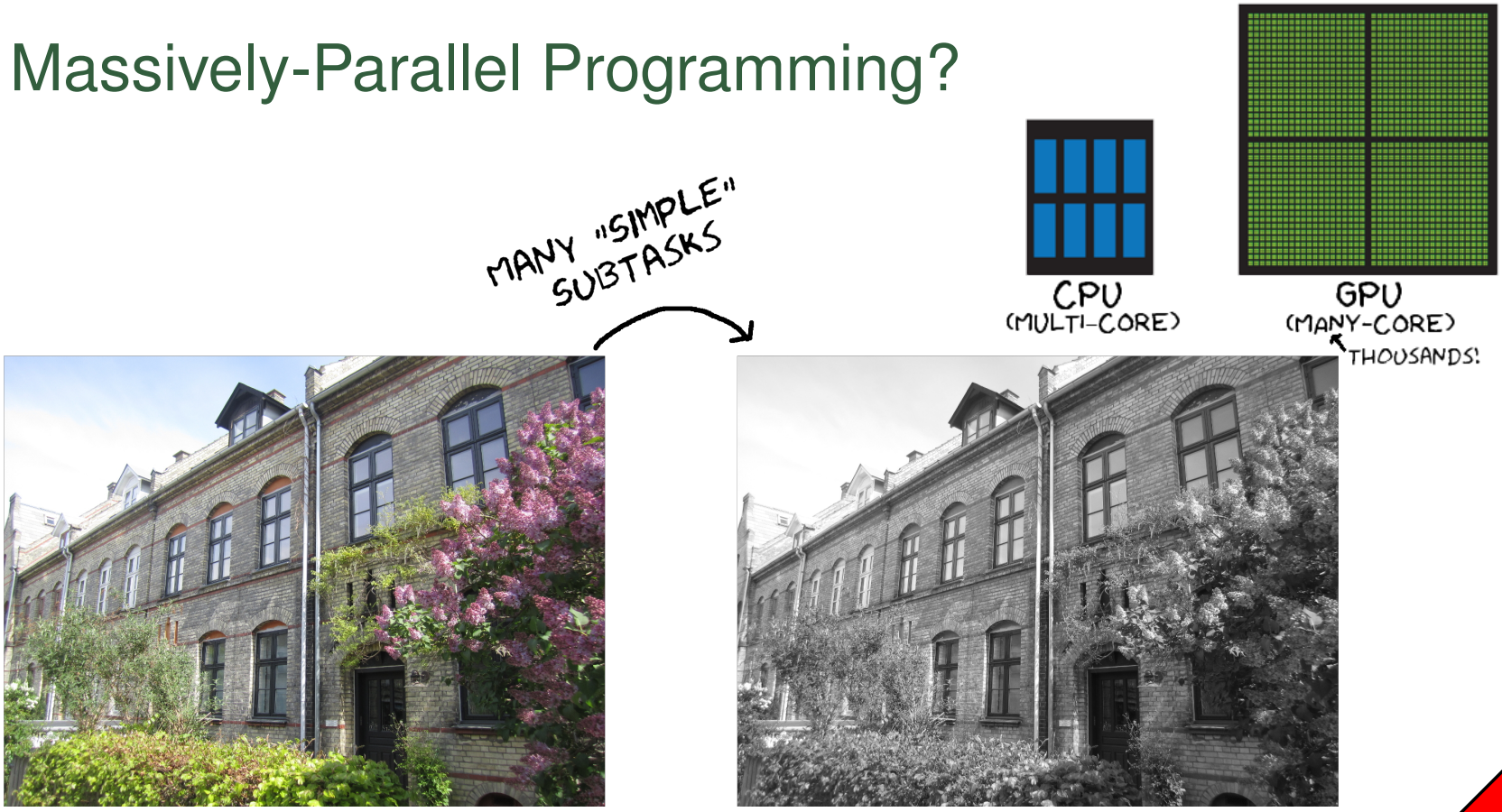


Graphics Processing Units (GPUs)

Can nowadays also be used for general computations and are well-suited for massively-parallel programming. Example: Adding two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{10000}$

1. CPU: Computes $x_1 + y_1$, $x_2 + y_2$, ... (sequentially)
2. GPU: Core i computes $x_i + y_i$ (in parallel)

Massively-Parallel Programming?



Graphics Processing Units (GPUs)

Can nowadays also be used for general computations and are well-suited for massively-parallel programming. Example: Adding two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{10000}$

1. CPU: Computes $x_1 + y_1$, $x_2 + y_2$, ... (sequentially)
2. GPU: Core i computes $x_i + y_i$ (in parallel)

Challenge
Adapt approach $\rightarrow$ Massively parallel?

General-Purpose Computations on GPUs

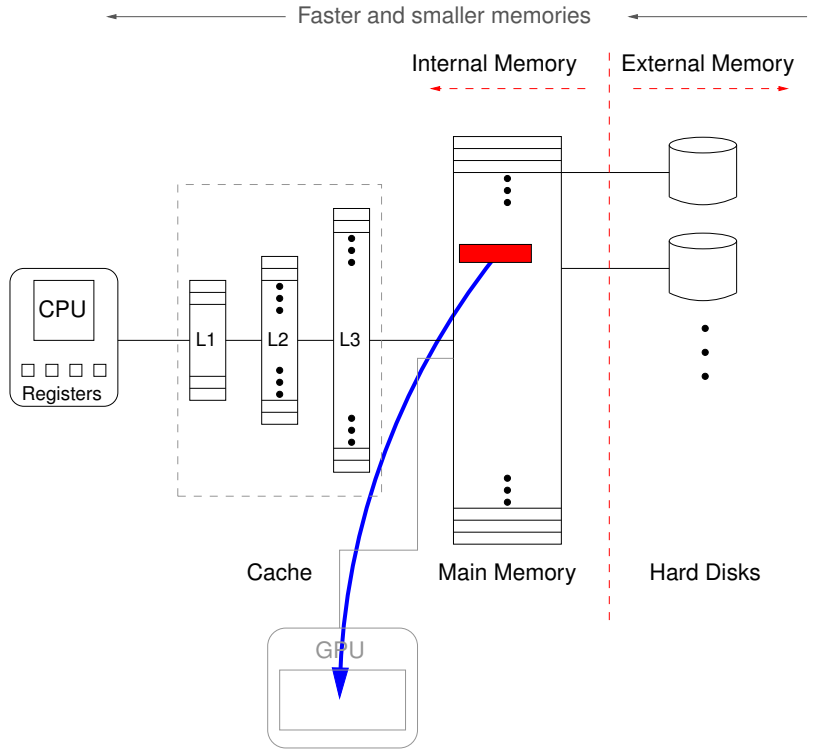
Multiple Instruction, Multiple Data (MIMD)

Each thread can execute a different instruction on a different piece of data.

Single Instruction, Multiple Data (SIMD)

A group of threads executes the same operation on different pieces of data.

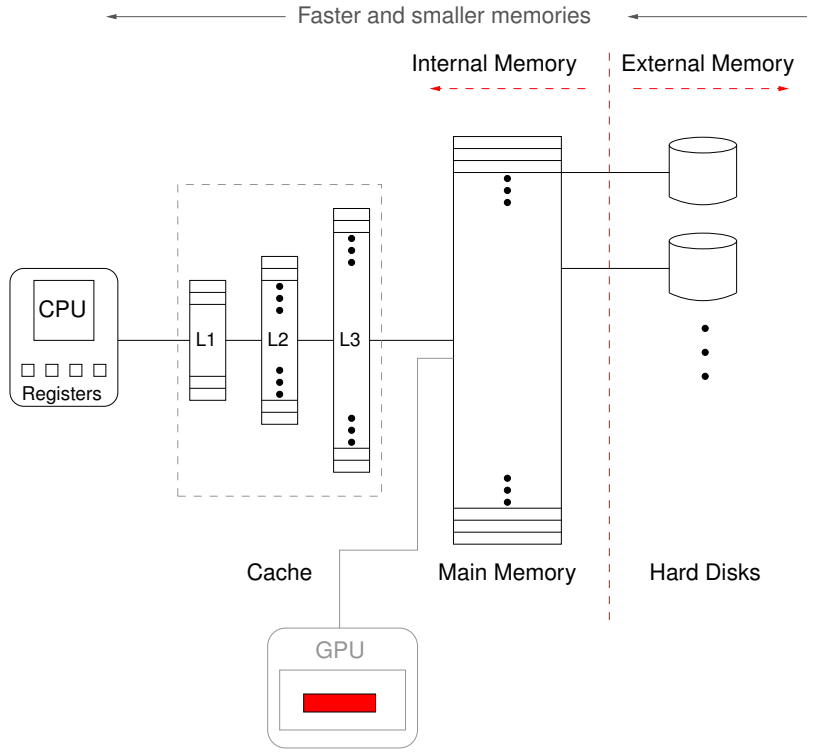
GPGPU Programming



Workflow

1. Copy data from main memory (host) to the main memory of the GPU.

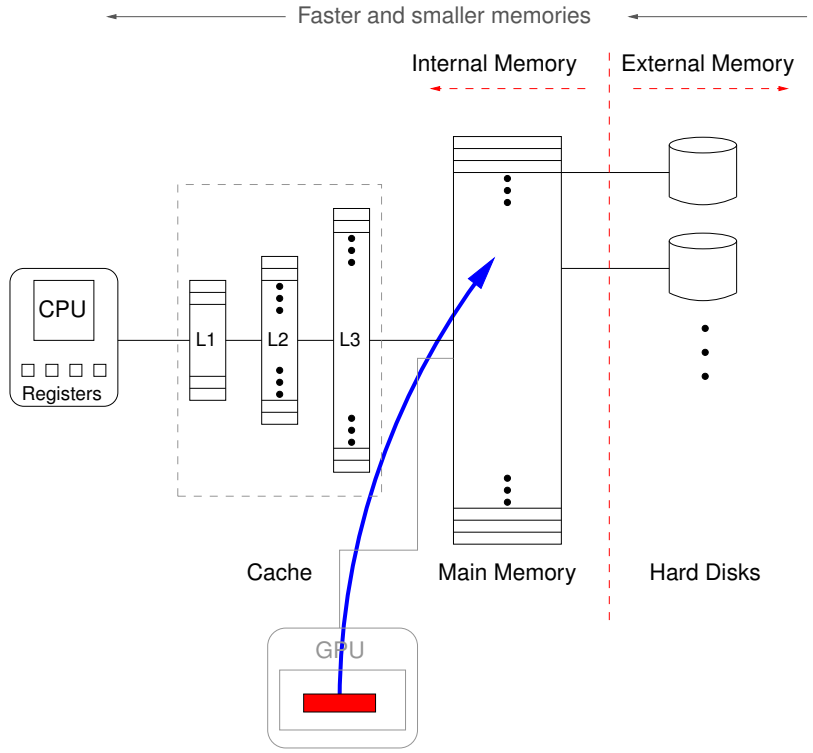
GPGPU Programming



Workflow

1. Copy data from main memory (host) to the main memory of the GPU.
2. Load and execute GPU program. Such a program is called kernel and is executed in a massively-parallel fashion.

GPGPU Programming



Workflow

1. Copy data from main memory (host) to the main memory of the GPU.
2. Load and execute GPU program. Such a program is called kernel and is executed in a massively-parallel fashion.
3. Once computations are done, copy the result back to host main memory.

Brute-Force + GPUs

formed in the distance matrix as part of the sorting processes were simultaneously applied to the index matrix so that, in the end, its uppermost $k \times n$-submatrix corresponded to the queries k-nearest neighbor indices ordered by increasing distance.
Working with an initial $m \times n$-index matrix represents a waste of memory. A simple "trick" allows us to work with a $k \times n$-index matrix from the beginning, thus avoiding the $(m - k) \times n$ memory overhead. The sorting process deals with each array column *monotonically* from the first to the last element. Consequently, at each iteration, the index of the considered reference point is known. While sorting, if the l-th element needs to be inserted into the distance matrix, the index value l is also inserted into the $k \times n$-index matrix at the exact same position, iteratively filling in the whole matrix.

The main result of [8] is that the proposed CUDA implementation was up to 300X faster than a similar C implementation and up to 150X faster than the highly optimized ANN C++ library [4].

2.3. CUBLAS implementation

BLAS is the celebrated, highly optimized linear algebra library specialized in vector/matrix operations. CUBLAS is the CUDA implementation of BLAS and improves the performance of the classical BLAS functions. The CUDA implementation of the kNN search [8] was very efficient in terms of computation time. The distance matrix computation represented the main part of the computation time. In this paper, we show that we can greatly improve the global performances of the kNN search by reformulating the way to compute the distance matrix and by using the CUBLAS library.

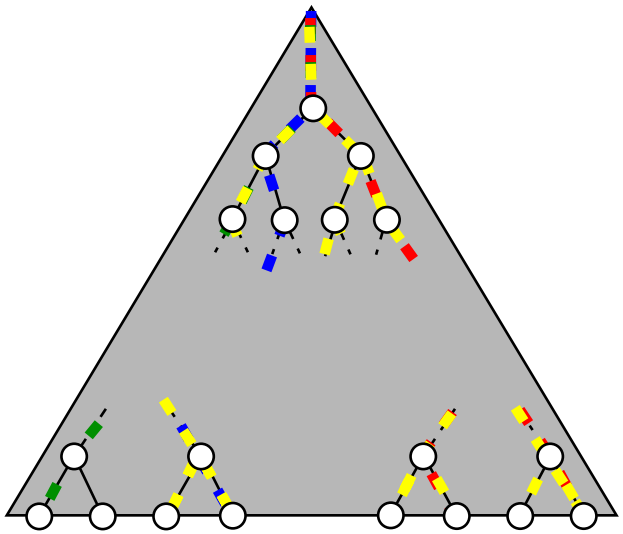
The proposed kNN search implementation is then based on CUDA and CUBLAS and is composed of the following kernels:

1. Compute the vector N_R using CUDA (coalesced read/write);
2. Compute the vector N_Q using CUDA (coalesced read/write);
3. Compute the $m \times n$-matrix $A = -2R^\top Q$ using CUBLAS;
4. Add the i^{th} element of N_R to every element of the i^{th} row of the matrix A using CUDA (grid of $m \times n$ threads, non coalesced read/write: use of the shared memory); The resulting matrix is denoted by B;
5. Sort in parallel each column of B (with n threads) using the modified insertion sort proposed in [8]; The resulting matrix is denoted by C;
6. Add the j^{th} value of N_Q to the first k elements of the j^{th} column of the matrix C using CUDA (coalesced read/write); The resulting matrix is denoted by D;
7. Compute the square root of the first k elements of D to obtain the k smallest distances (coalesced read/write); The resulting matrix is denoted by E;
8. Extract the uppermost $k \times n$-submatrix of E; The resulting matrix is the desired distance matrix for the k-nearest neighbors of each query.

Note that the matrix names were given for algorithmic clarity only. Actually, once A is computed, all the remaining computations are done "in place", meaning that the matrices from A to E are in fact a

Garcia, Debreuve, Nielsen, Barlaud. *K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching*. IEEE International Conference on Image Processing, 2010.

GPUs and *k*-d Trees?



Multiple Instruction, Multiple Data (MIMD)

A typical parallel *k*-d tree implementation assigns one thread to each query and all threads traverse the tree simultaneously.

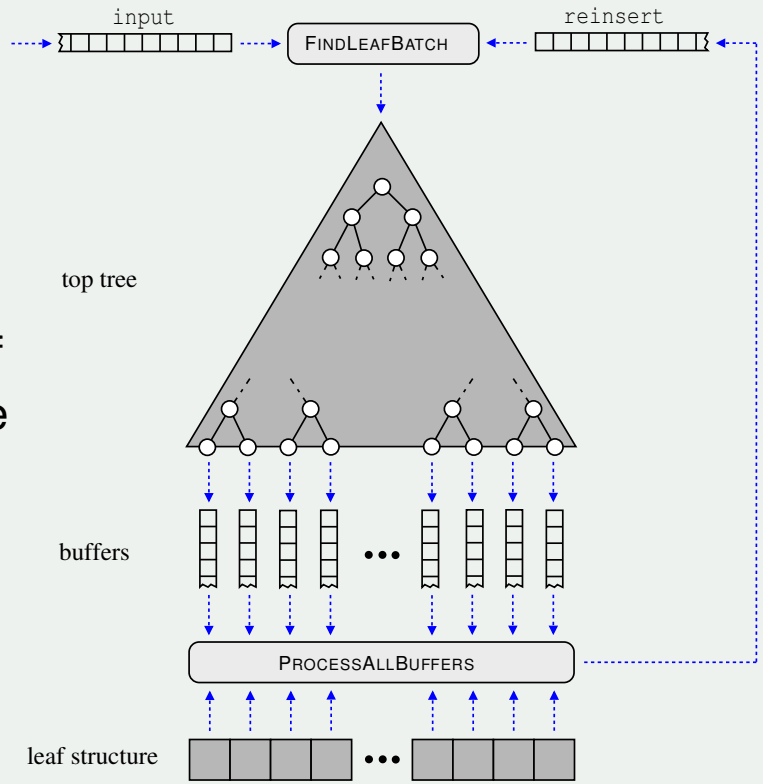
Single Instruction, Multiple Data (SIMD)

Each query may induce a completely different tree traversal $\rightarrow$ branch divergence and irregular/expensive accesses to global memory $\rightarrow$ ill-suited!

Lazy Parallel Nearest Neighbor Search

Buffer k-d Trees

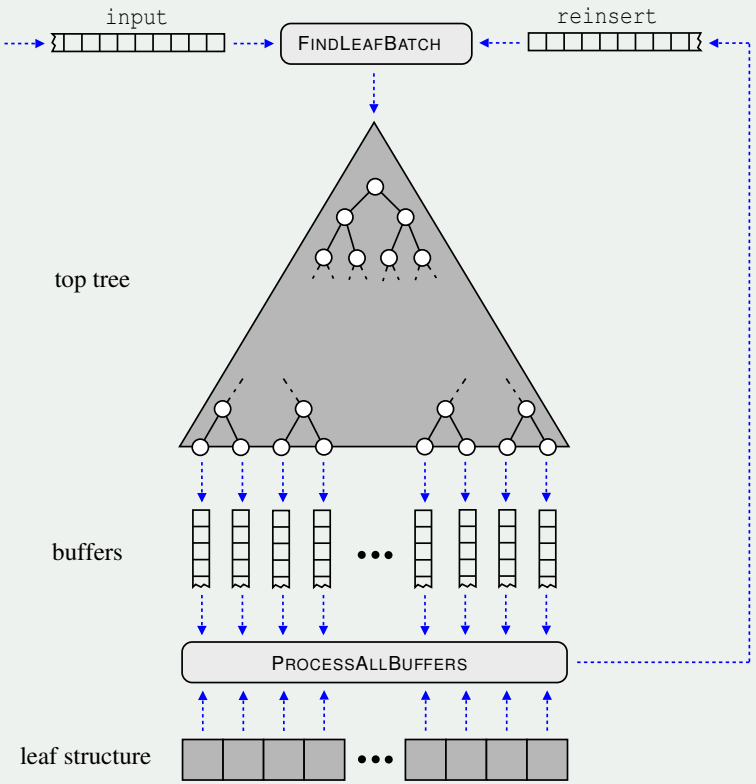
1. Top tree: First levels of a standard k-d tree (i.e., its median values), laid out in memory in a pointer-less manner.
2. Leaf structure: Training patterns, sorted *in-place* during the construction of the top tree (w.r.t. the median values). Each block of the leaf structure corresponds to a leaf of the top tree. [stored consecutively in memory]
3. Buffers: One buffer for each leaf of the top tree; each buffer can store a predefined number B of query indices.
4. Queues `input` and `reinsert`: Two (first-in-first-out) queues of size m.



Lazy Parallel Nearest Neighbor Search

Buffer *k*-d Trees

1. Top tree: First levels of a standard *k*-d tree (i.e., its median values), laid out in memory in a pointer-less manner.
2. Leaf structure: Training patterns, sorted *in-place* during the construction of the top tree (w.r.t. the median values). Each block of the leaf structure corresponds to a leaf of the top tree. [stored consecutively in memory]
3. Buffers: One buffer for each leaf of the top tree; each buffer can store a predefined number B of query indices.
4. Queues `input` and `reinsert`: Two (first-in-first-out) queues of size m.



Key Idea: Use buffer k-d tree to delay processing of queries.

Multi-Core vs. Many-Core

	psf_colors ($d=4$)	psf_mag ($d=5$)	psf_model_mag ($d=10$)	all_mag ($d=15$)	all_colors ($d=12$)	all ($d=27$)
kdtree(cpu,1)	338 (×24)	259 (×22)	1965 (×55)	12459 (×59)	42314 (×89)	-
kdtree(cpu,2)	173 (×12)	130 (×11)	1012 (×28)	6478 (×31)	22108 (×46)	-
kdtree(cpu,3)	117 (× 8)	91 (× 8)	707 (×20)	5059 (×24)	17724 (×37)	-
kdtree(cpu,4)	92 (× 7)	70 (× 6)	584 (×16)	4639 (×22)	16770 (×35)	-
kdtree(cpu,8)	71 (× 5)	57 (× 5)	527 (×15)	4616 (×22)	16394 (×34)	-
bruteforce(gpu)	552 (×39)	664 (×55)	1160 (×32)	1595 (× 8)	1586 (× 3)	3047 (×2)
kdtree(gpu)	73 (× 5)	67 (× 6)	445 (×12)	3050 (×15)	10373 (×22)	-
bufferkdtree(gpu)	14	12	36	210	478	1717

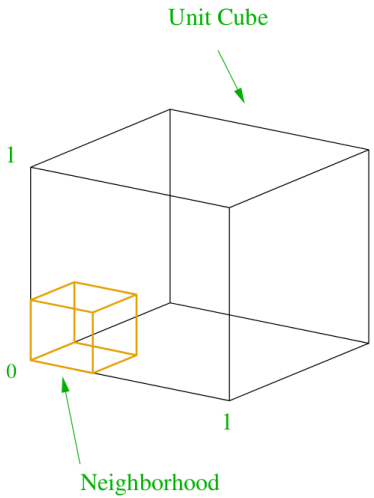
Table 1: Runtime comparison (in seconds) given $n = \#train = 2 \cdot 10^6$ and $m = \#test = 10^7$ ($k = 10$; the speed-up of bufferkdtree(gpu) over its competitors is given in brackets). For all tree-based methods, optimal tree heights were chosen in a preprocessing phase.

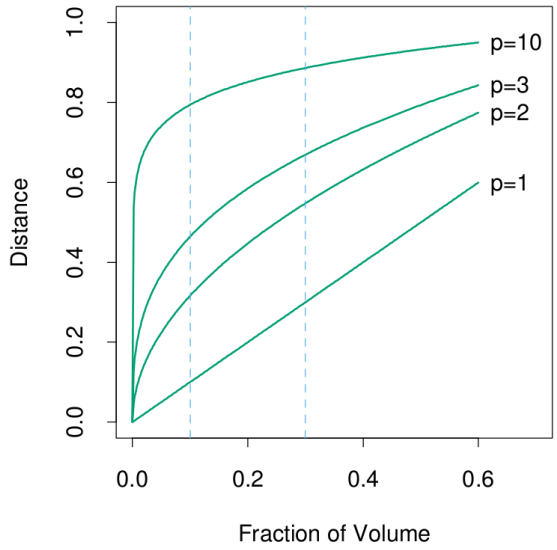
Intel i7@3.40GHz (4 cores, 8 hard. threads), GeForce GTX 770 (1536 cores, 4GB RAM)

Gieseke, Heinermann, Oancea, Igel. *Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs.* ICML, 2014.

Outline

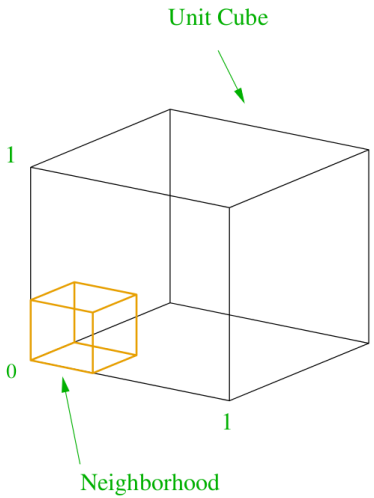
Nearest Neighbors in High Dimensions?

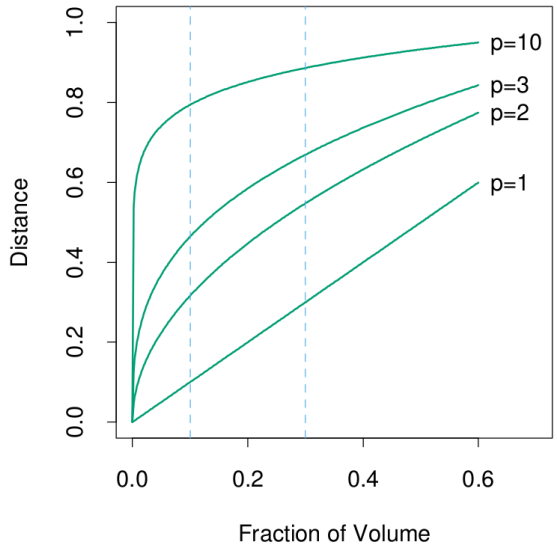




Example: Side-length of the subcube needed to capture a fraction r of the volume of the data for different dimensions p. For $p = 10$, we need 80% of the range of each coordinate to cover 10% of the data, see [HTF09] for details.

Nearest Neighbors in High Dimensions?



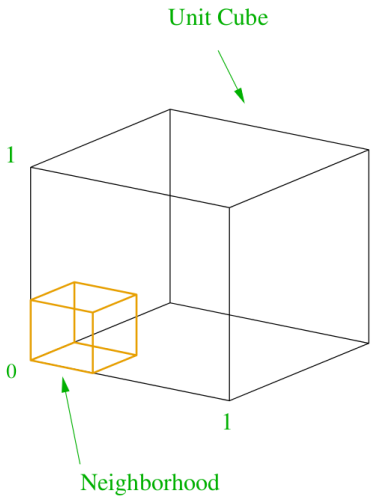


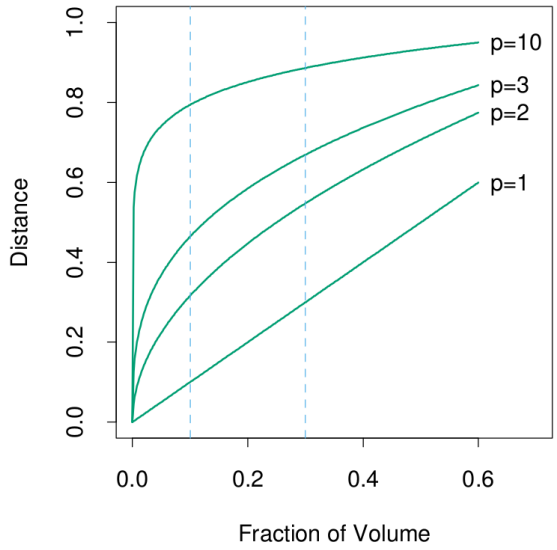
Example: Side-length of the subcube needed to capture a fraction r of the volume of the data for different dimensions p. For $p = 10$, we need 80% of the range of each coordinate to cover 10% of the data, see [HTF09] for details.

Curse of Dimensionality

The curse of dimensionality describes, among other things, the fact that we might need an “exponential amount” of training data.

Nearest Neighbors in High Dimensions?





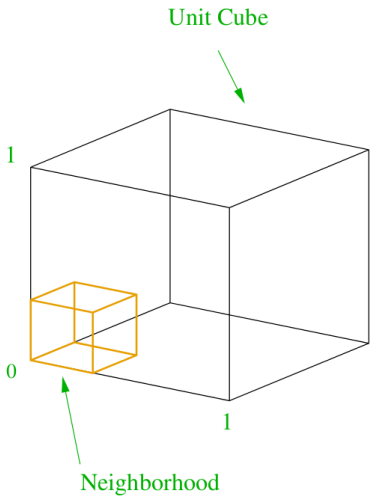
Example: Side-length of the subcube needed to capture a fraction r of the volume of the data for different dimensions p. For $p = 10$, we need 80% of the range of each coordinate to cover 10% of the data, see [HTF09] for details.

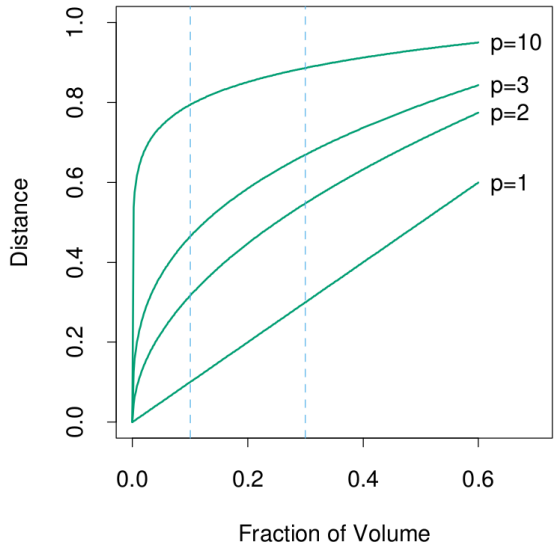
Curse of Dimensionality

The curse of dimensionality describes, among other things, the fact that we might need an "exponential amount" of training data.

1. Solution 1: Process a lot (!) more training instances!

Nearest Neighbors in High Dimensions?





Example: Side-length of the subcube needed to capture a fraction r of the volume of the data for different dimensions p. For $p = 10$, we need 80% of the range of each coordinate to cover 10% of the data, see [HTF09] for details.

Curse of Dimensionality

The curse of dimensionality describes, among other things, the fact that we might need an "exponential amount" of training data.

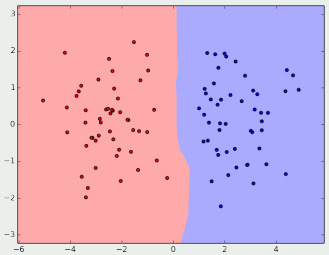
1. Solution 1: Process a lot (!) more training instances!
2. Solution 2: "Reduce" the dimensionality of the feature space!

Noisy Features in High Dimensions

Example

Two Gaussians in high dimensions: The clusters are generated via $X_i \sim \mathcal{N}(\mathbf{m}_i, \mathbf{I})$ with $\mathbf{m}_1 = (-2.5, 0.0, \ldots, 0.0)^\top \in \mathbb{R}^d$ and $\mathbf{m}_2 = (+2.5, 0.0, \ldots, 0.0)^\top \in \mathbb{R}^d$.

- Model: Nearest neighbors with $k = 5$
- Question: What happens if we
  1. increase the dimensionality (fixed # of patterns)?
  2. the number of patterns (fixed dimension)?

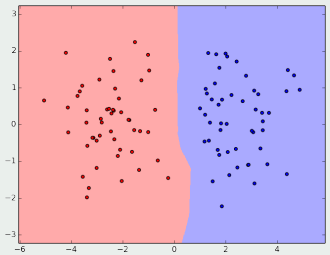


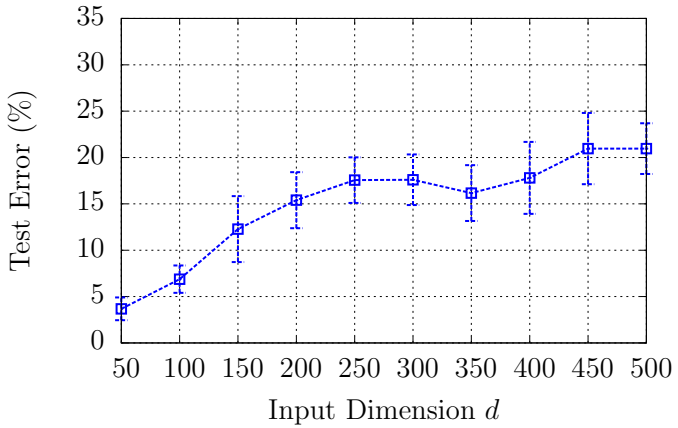
Noisy Features in High Dimensions

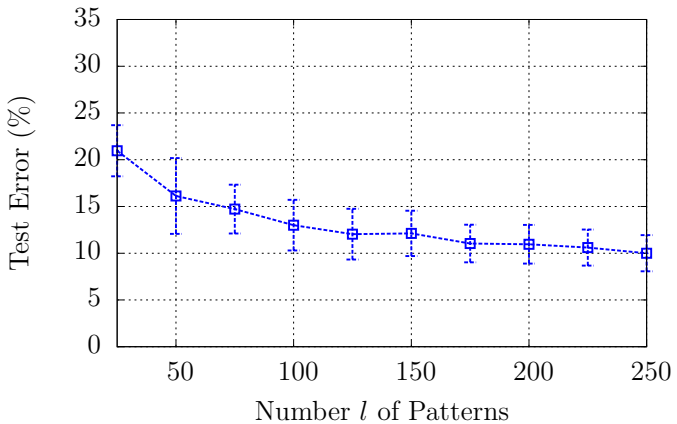
Example

Two Gaussians in high dimensions: The clusters are generated via $X_i \sim \mathcal{N}(\mathbf{m}_i, \mathbf{I})$ with $\mathbf{m}_1 = (-2.5, 0.0, \ldots, 0.0)^\top \in \mathbb{R}^d$ and $\mathbf{m}_2 = (+2.5, 0.0, \ldots, 0.0)^\top \in \mathbb{R}^d$.

- Model: Nearest neighbors with $k = 5$
- Question: What happens if we
  1. increase the dimensionality (fixed # of patterns)?
  2. the number of patterns (fixed dimension)?







Left: 25 training patterns. Right: Fixed dimension $d = 500$.

Dimensionality Reduction

Two approaches to reducing the dimensionality of the feature space are:

1. Feature extraction: Transformation of a feature space into a lower-dimensional space.
2. Feature selection: Selecting a subset of “good” features.

Dimensionality Reduction

Two approaches to reducing the dimensionality of the feature space are:

1. Feature extraction: Transformation of a feature space into a lower-dimensional space.
2. Feature selection: Selecting a subset of "good" features.

Feature Selection $\neq$ Feature Extraction

Feature extraction generates new features. In constrast, feature selection yields a subset of the original features (and we can still interpret them!).

Dimensionality Reduction

Two approaches to reducing the dimensionality of the feature space are:

1. Feature extraction: Transformation of a feature space into a lower-dimensional space.
2. Feature selection: Selecting a subset of "good" features.

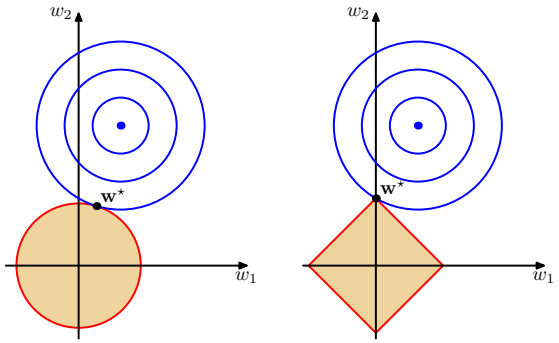
Feature Selection $\neq$ Feature Extraction

Feature extraction generates new features. In constrast, feature selection yields a subset of the original features (and we can still interpret them!).

Some of the potential benefits of feature selection are:

- We might get rid of "unimportant" features or noise.
- Features might be expensive to obtain $\rightarrow$ saving time/costs.
  (e.g., we might need sensors to generate them)
- Less storage requirements. Also, we might need less computational resources in the training and testing phase!
  (k-d trees might work much better!)

Embedded and Wrapper Methods



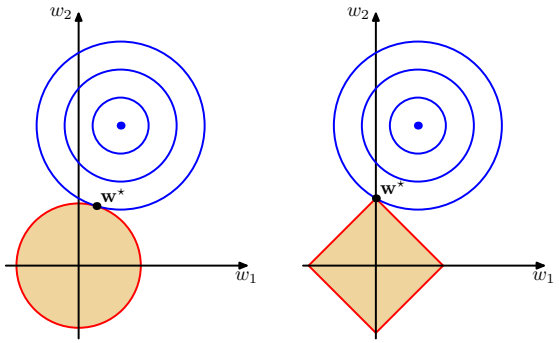
(a) Embedded methods: Regularization can be used to select important features:

$$\underset{\mathbf{w}}{\text{minimize}} \sum_{i=1}^{n} \left(y_i - w_0 - \sum_{j=1}^{p} w_j x_{ij} \right)^2 + \lambda \|\mathbf{w}\|^q$$

where $q = 1$ or $q = 2$. By minimizing $\|\mathbf{w}\|^q$, we prefer small weights. For $p = 1$, one can show that this yields only few "active" non-zero weights that correspond to "important" features (lasso method, see also [HTF09]).

[Bis07]

Embedded and Wrapper Methods



(a) Embedded methods: Regularization can be used to select important features:

$$\underset{\mathbf{w}}{\text{minimize}} \sum_{i=1}^{n} \left(y_i - w_0 - \sum_{j=1}^{p} w_j x_{ij} \right)^2 + \lambda \|\mathbf{w}\|^q$$

where $q = 1$ or $q = 2$. By minimizing $\|\mathbf{w}\|^q$, we prefer small weights. For $p = 1$, one can show that this yields only few "active" non-zero weights that correspond to "important" features (lasso method, see also [HTF09]).

(b) Wrapper methods: Such schemes utilize the models as black box to score subsets of features according to their predictive power.

"Performance assessments are usually done using a validation set or by cross-validation. [...] An exhaustive search can conceivably be performed, if the number of variables is not too large. But, the problem is known to be NP-hard and the search becomes quickly computationally intractable" [GE03].

[Bis07]

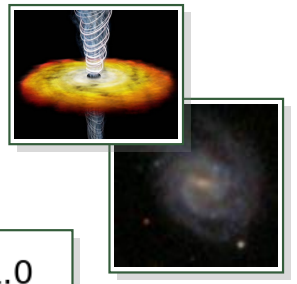
Alternatives: Forward and Backward Selection

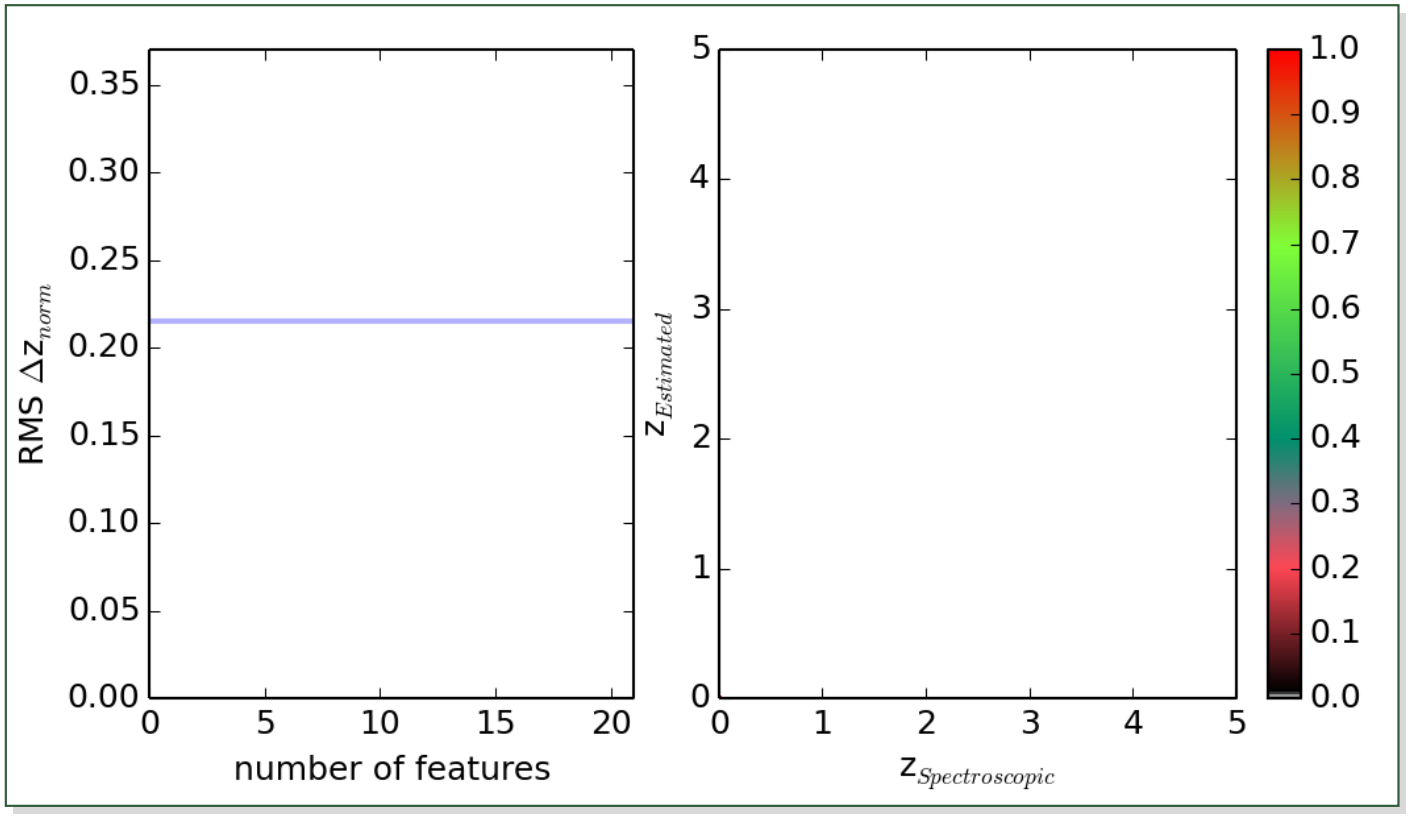
Selecting $f = 10$ out of $d = 585$ features requires checking $1,197,308,441,345,108,200,000$ combinations. We clearly need another strategy! Two well-known alternatives are

1. **Forward selection:** Start with an empty set and incrementally add the best-performing feature. Stop when f features are selected.
2. **Backward selection:** Start with all features and incrementally remove the worst feature. Stop when only f are remaining.

A variety of other schemes has been proposed in the literature including the combination of forward and backward selection, simulated annealing, genetic algorithms, and many others.

Example: Forward Feature Selection

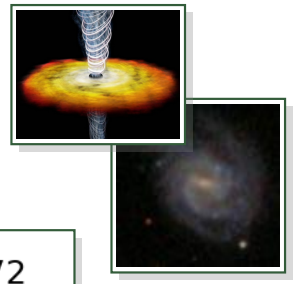


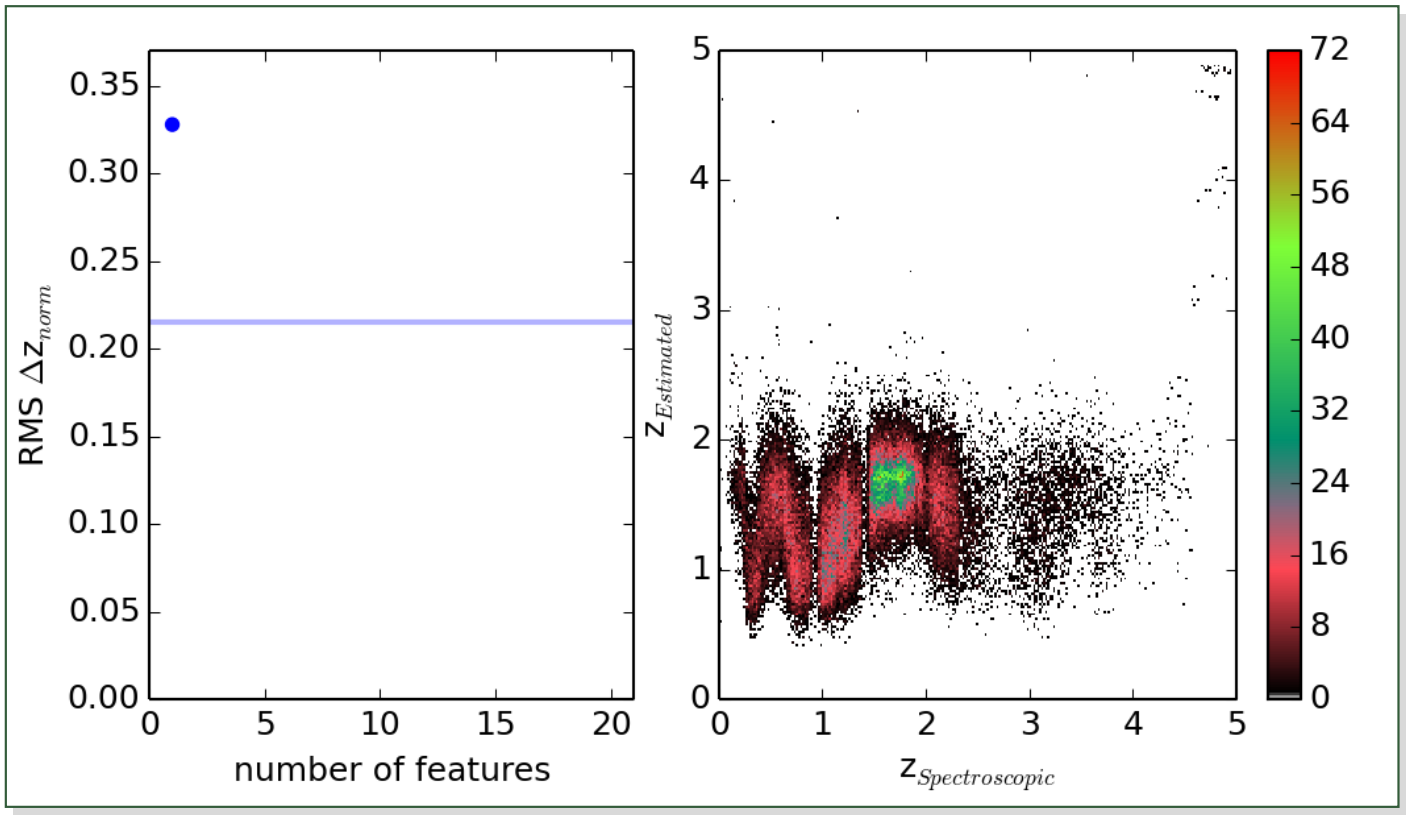


Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

Example: Forward Feature Selection

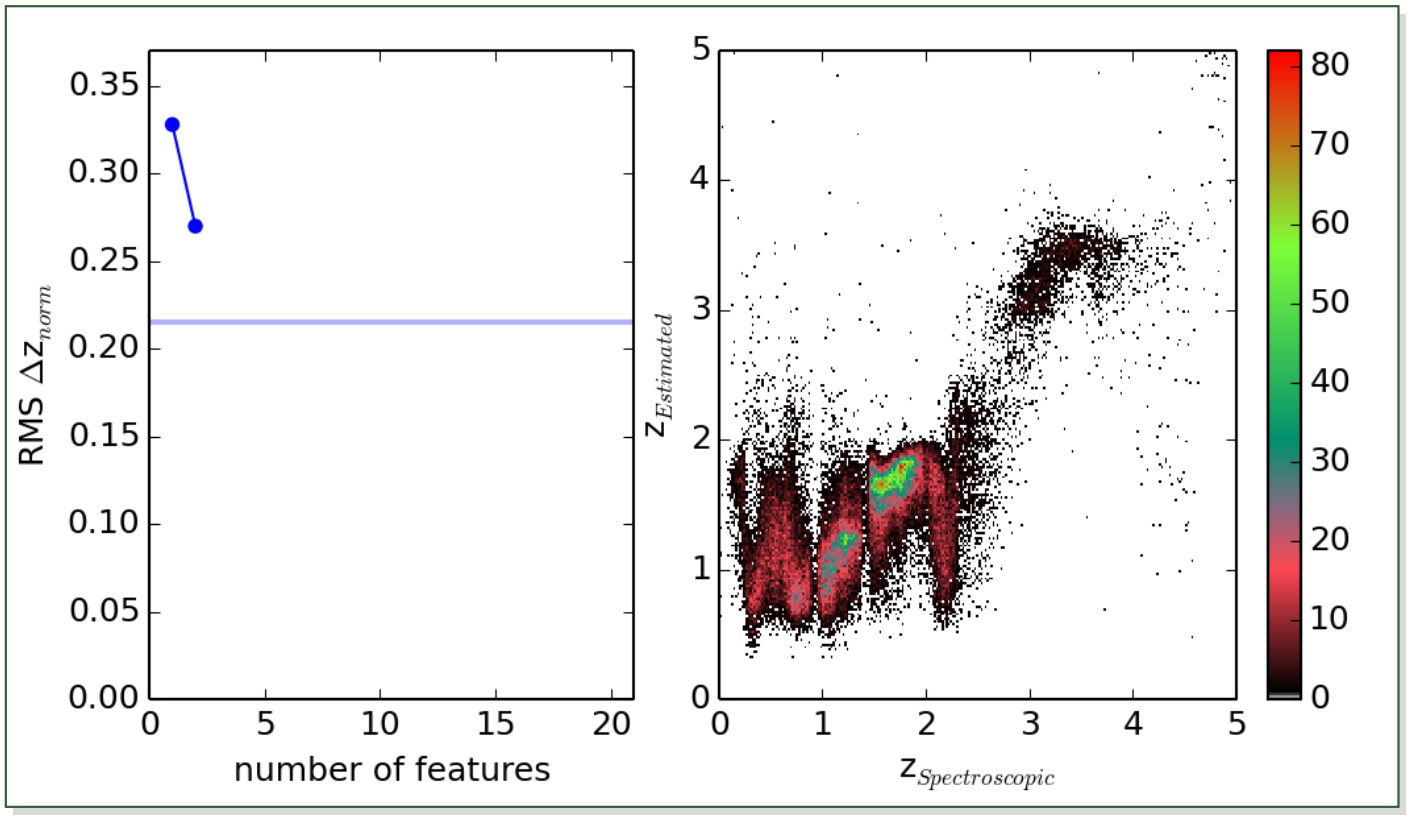




Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

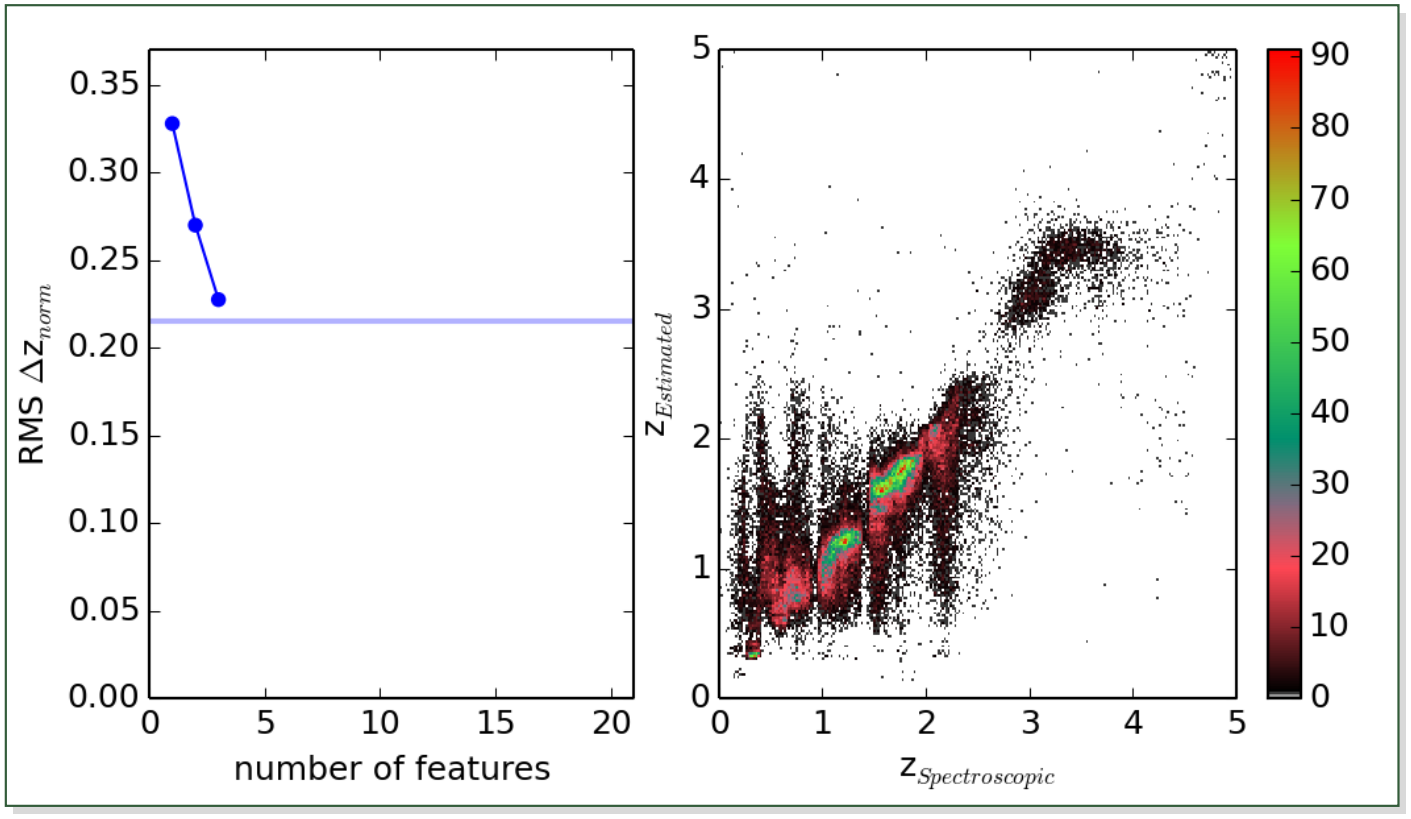
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

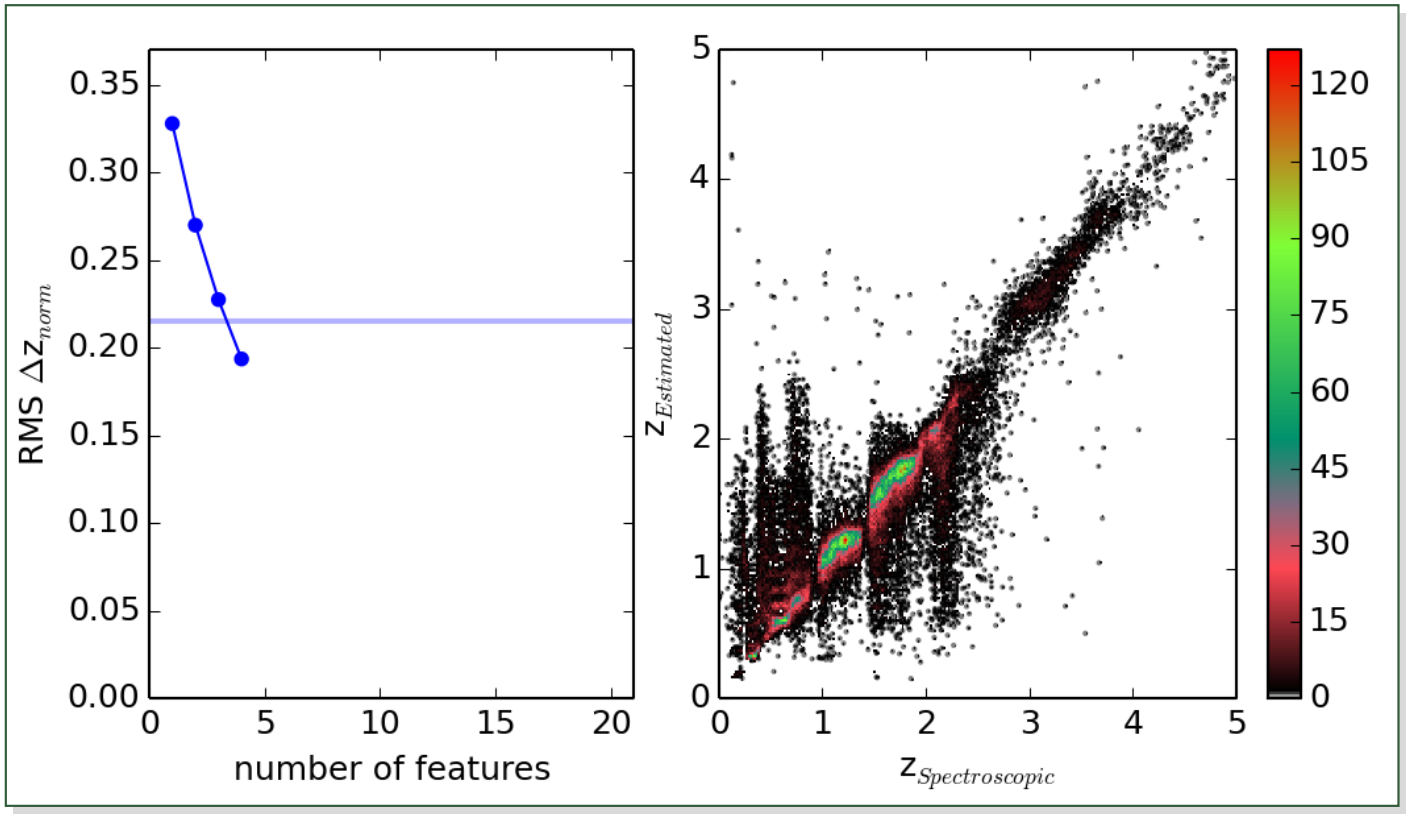
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

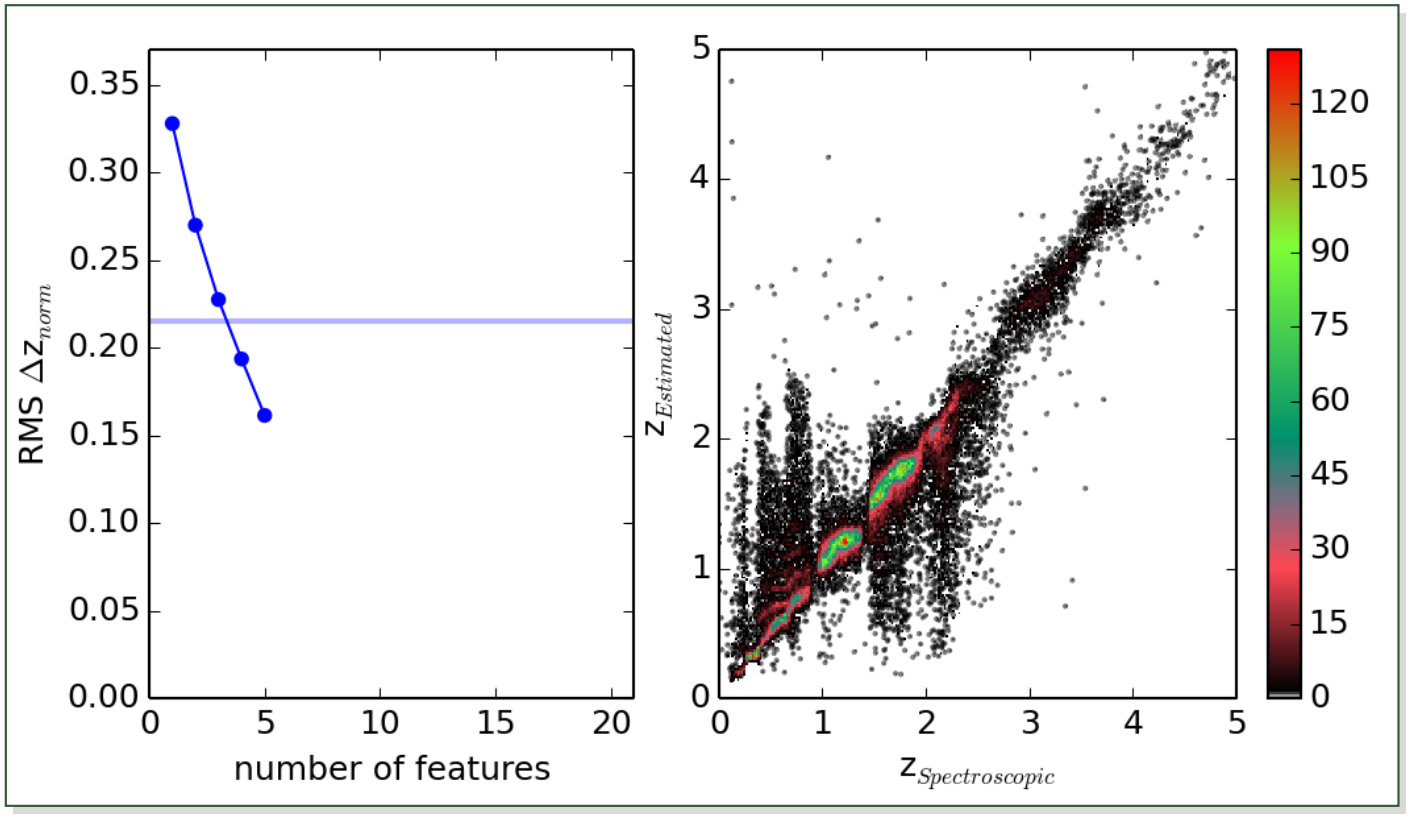
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

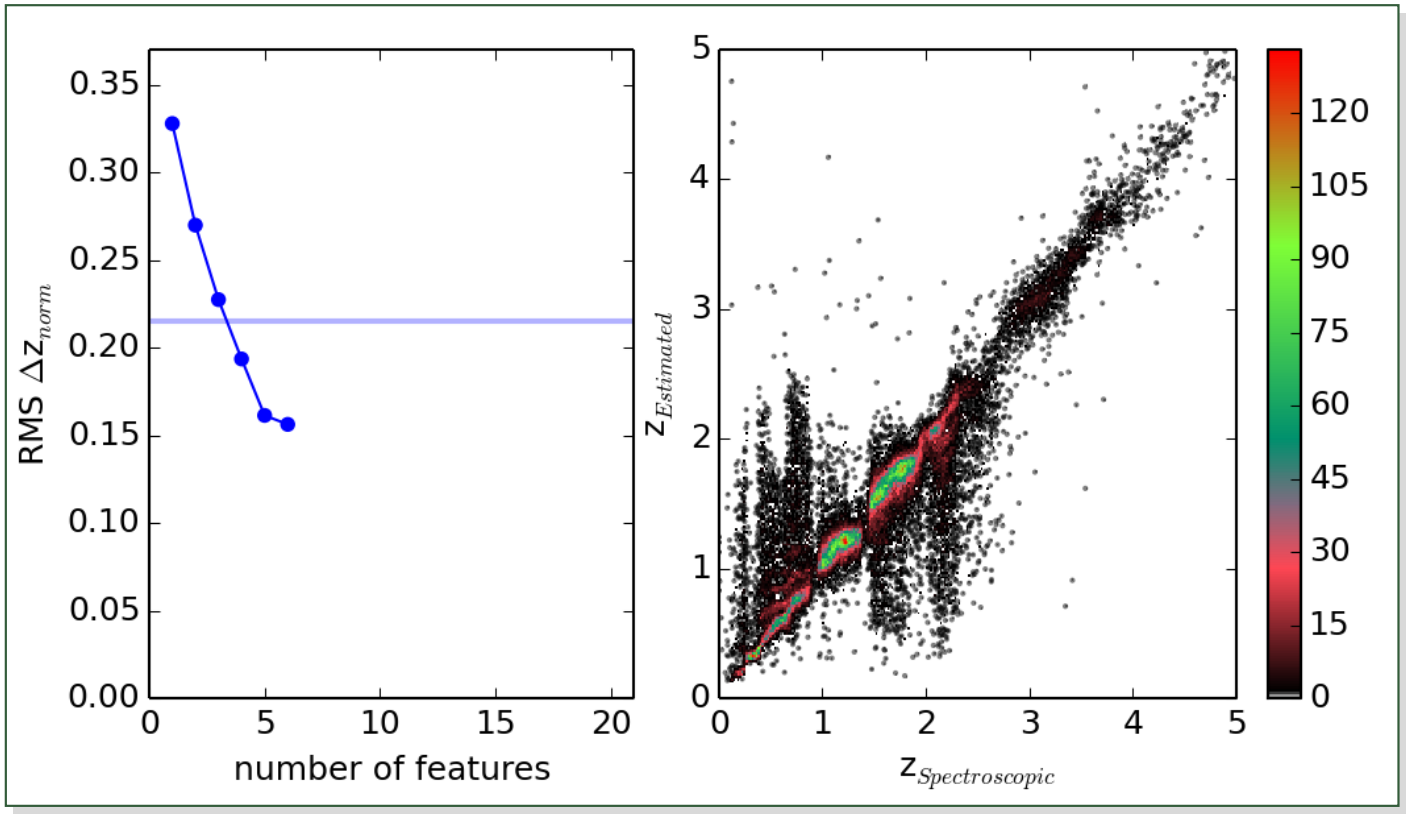
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

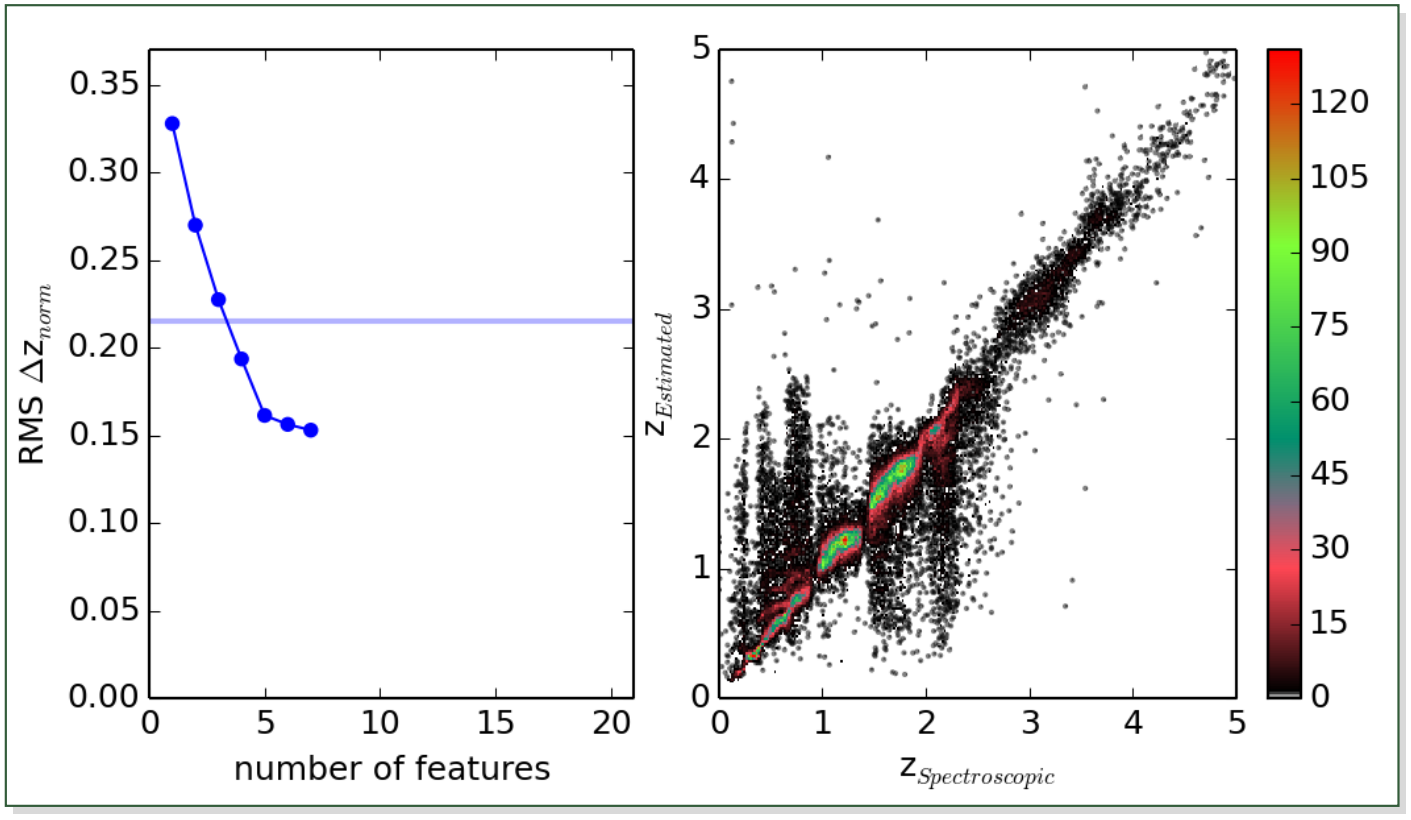
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

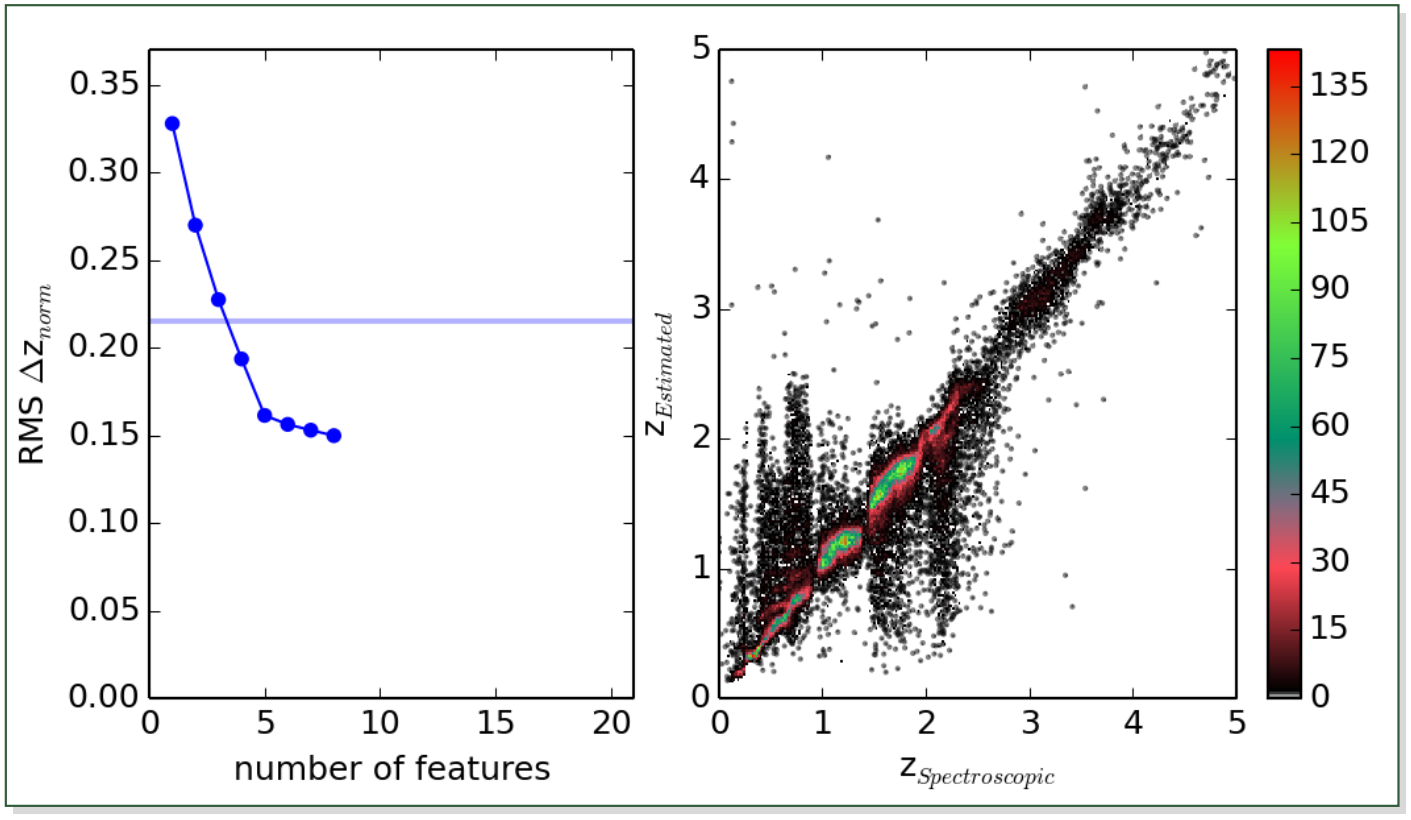
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

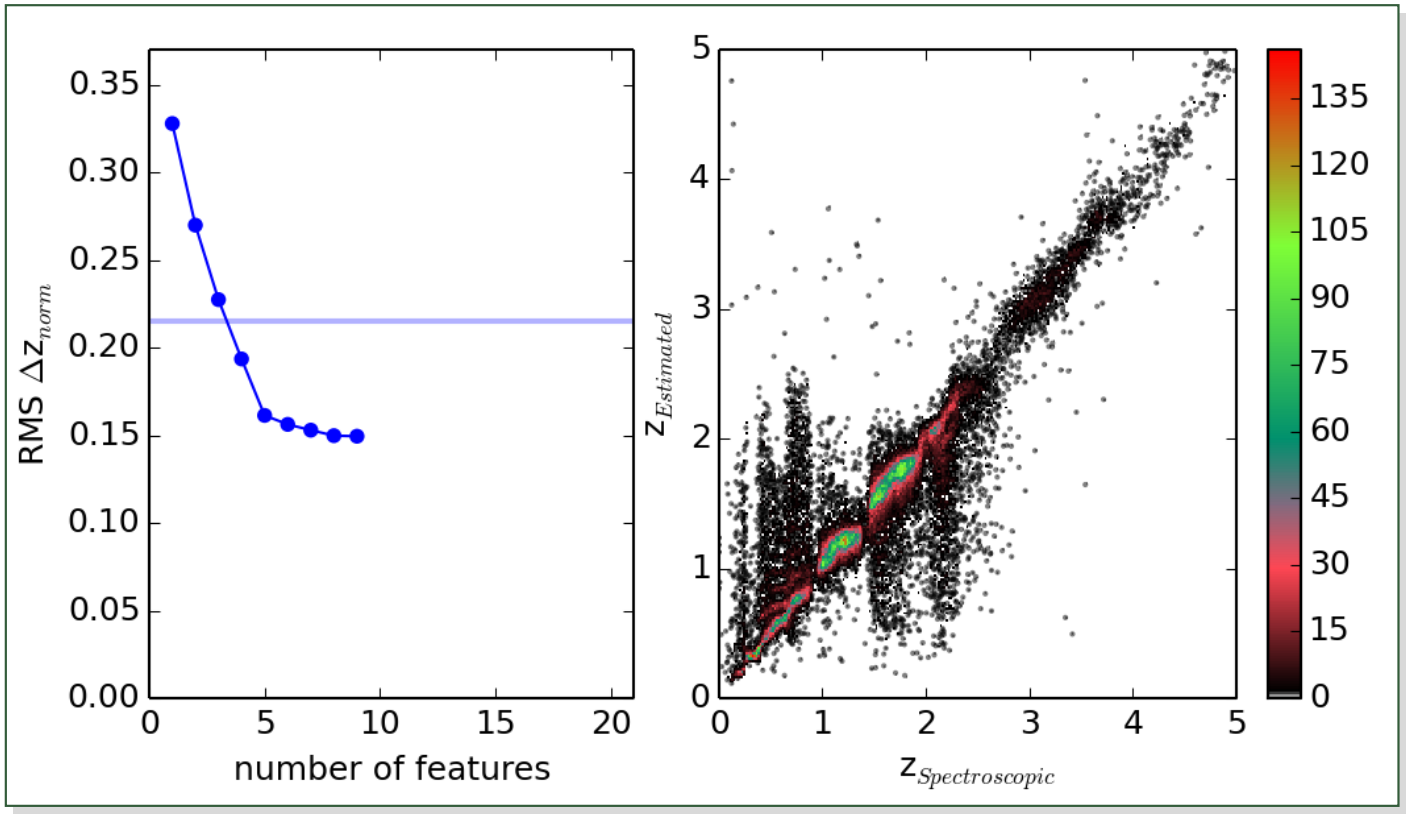
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

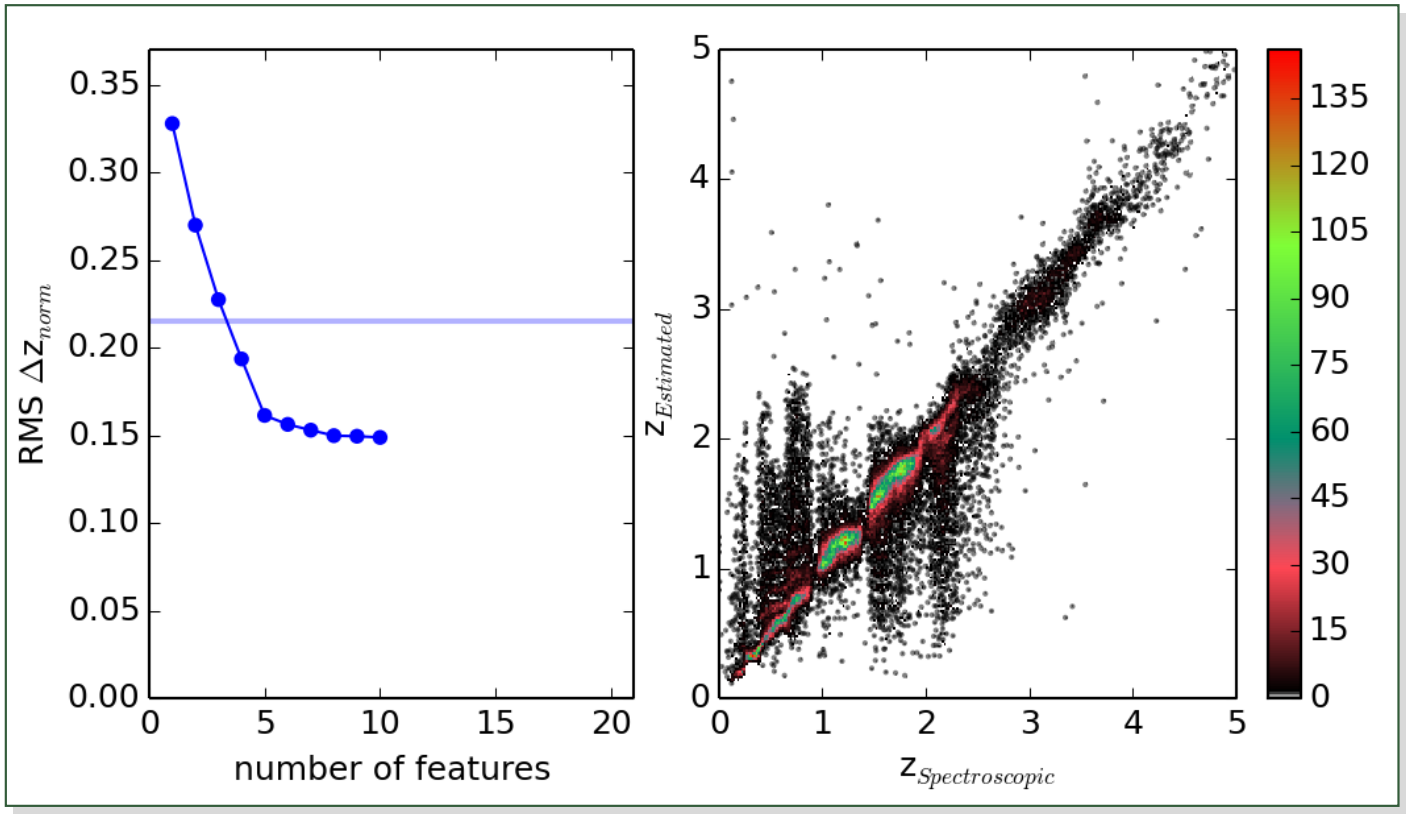
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

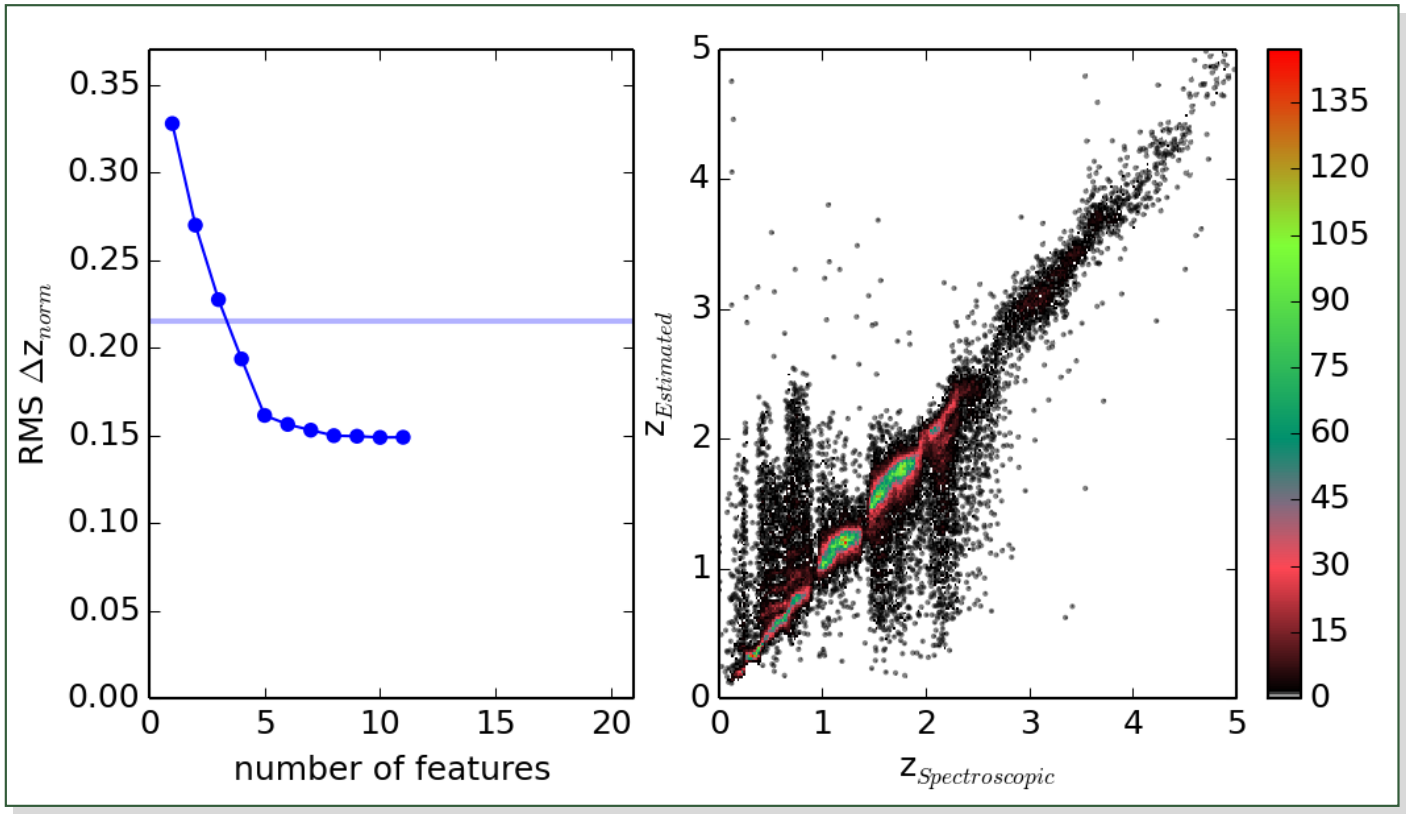
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

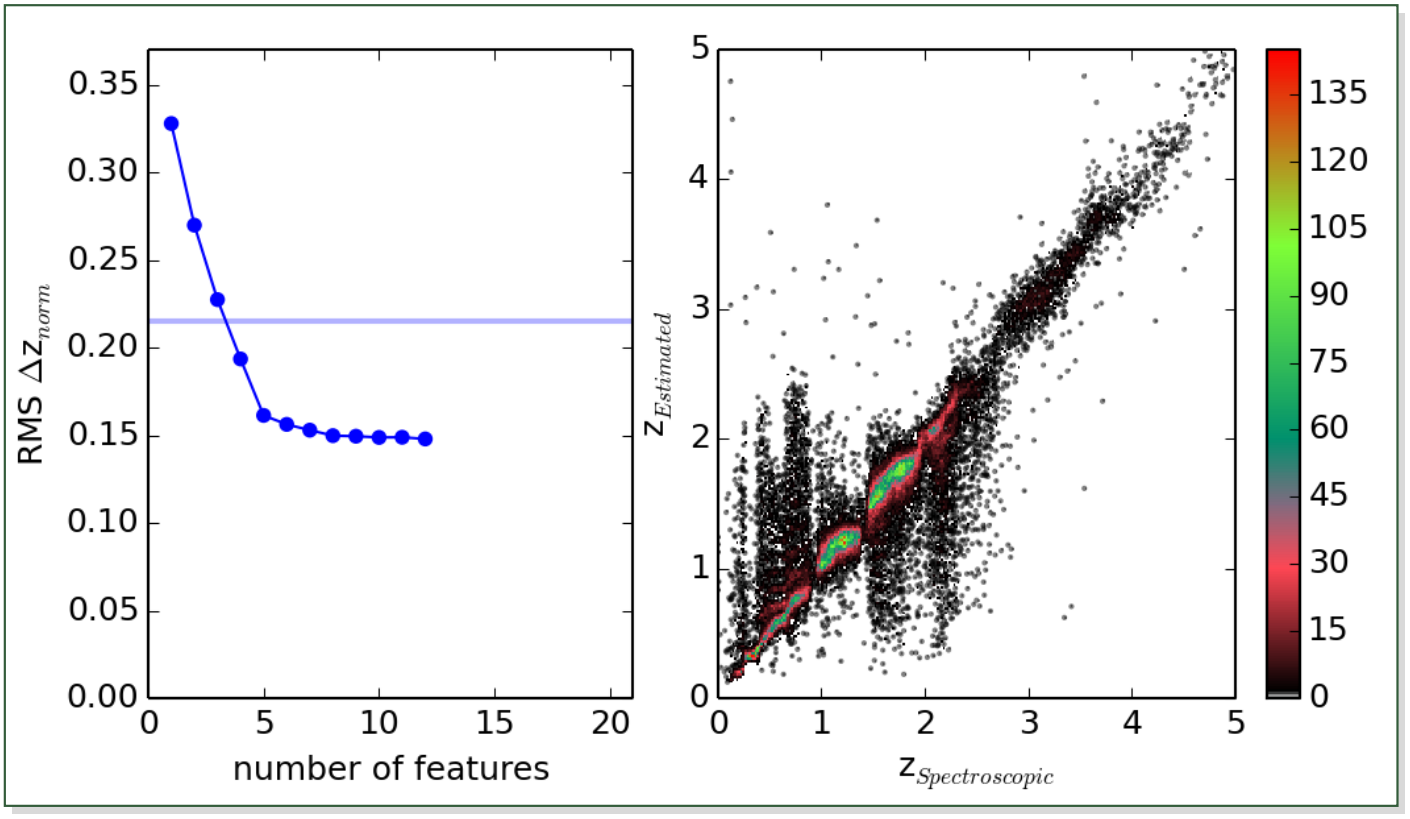
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

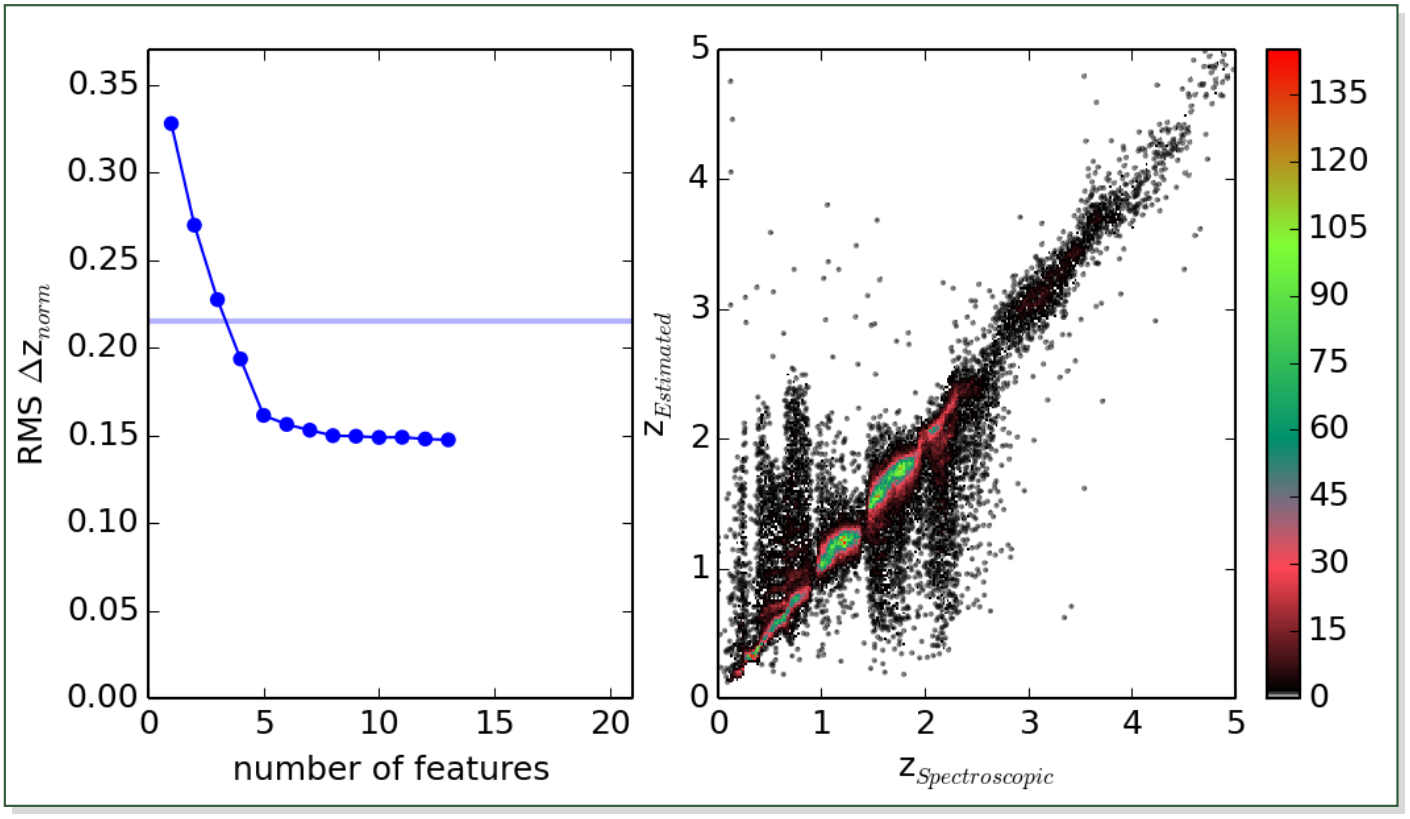
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

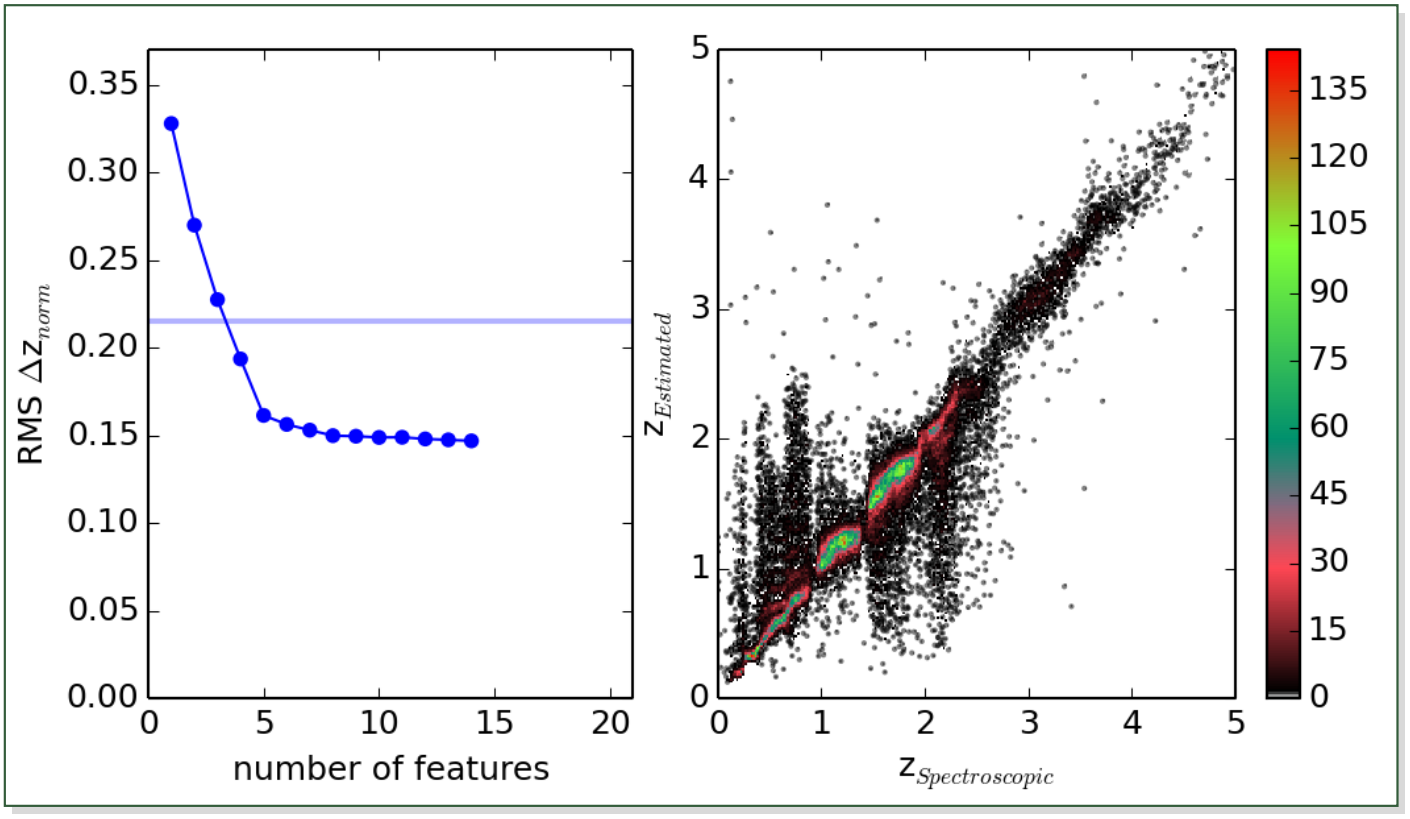
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

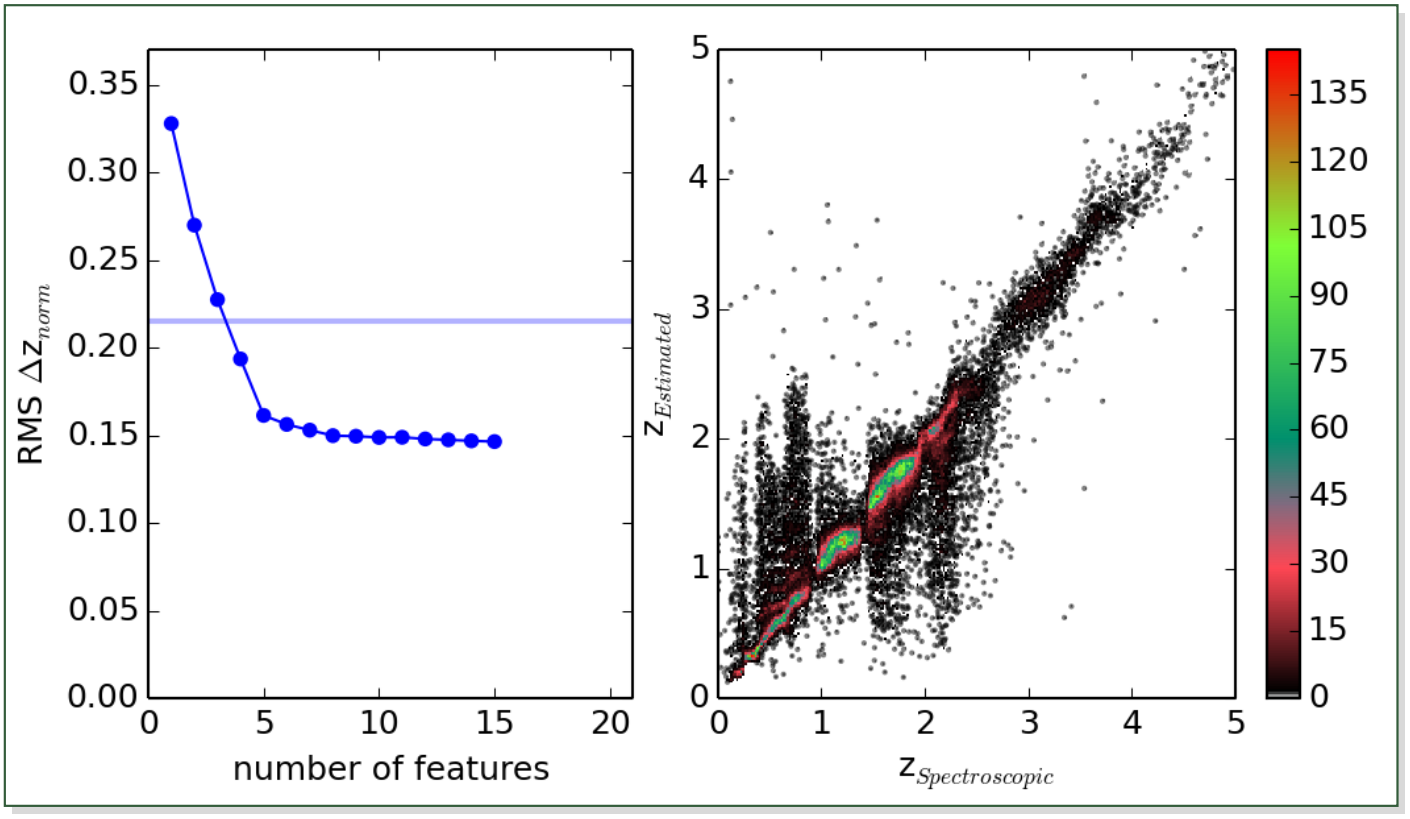
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

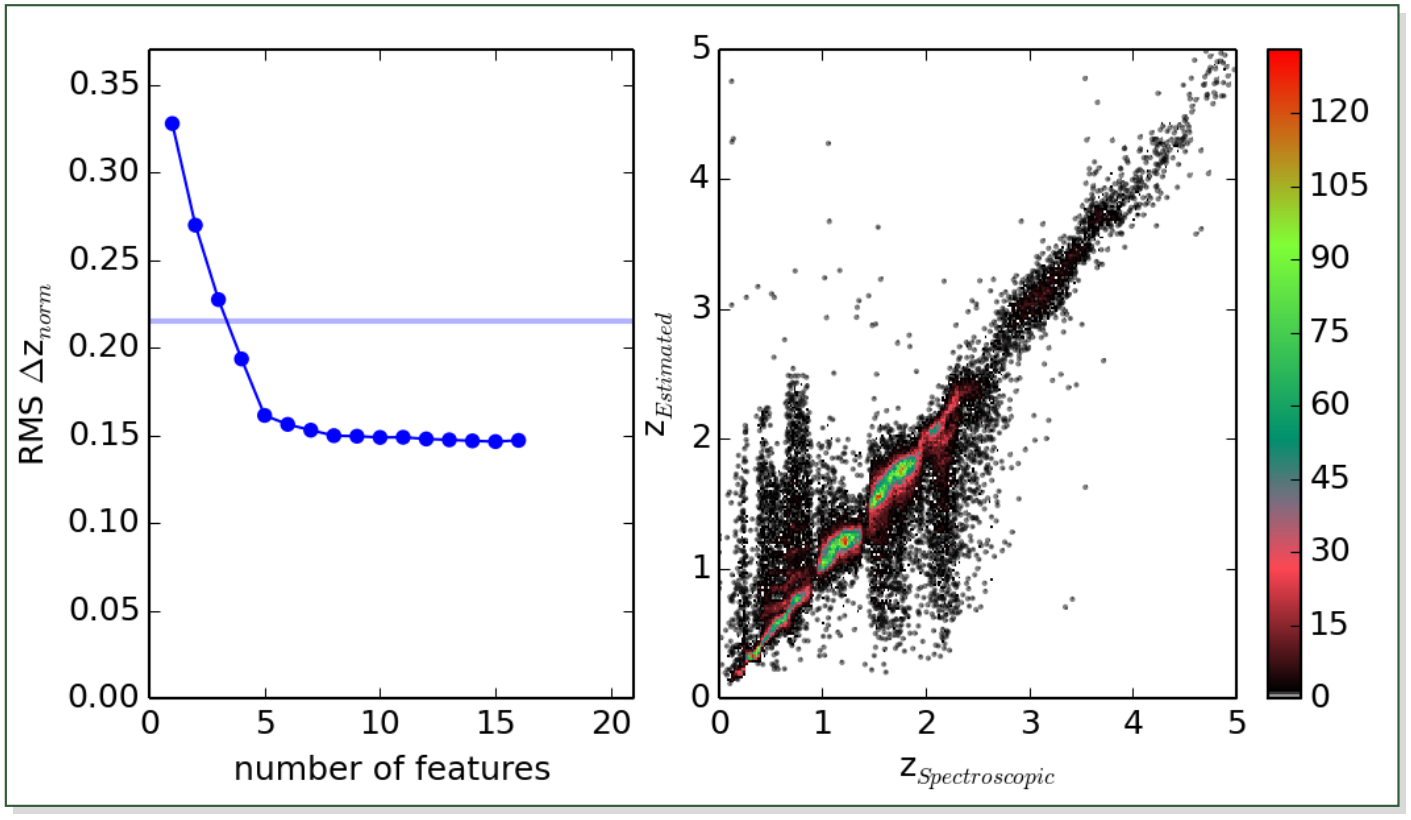
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

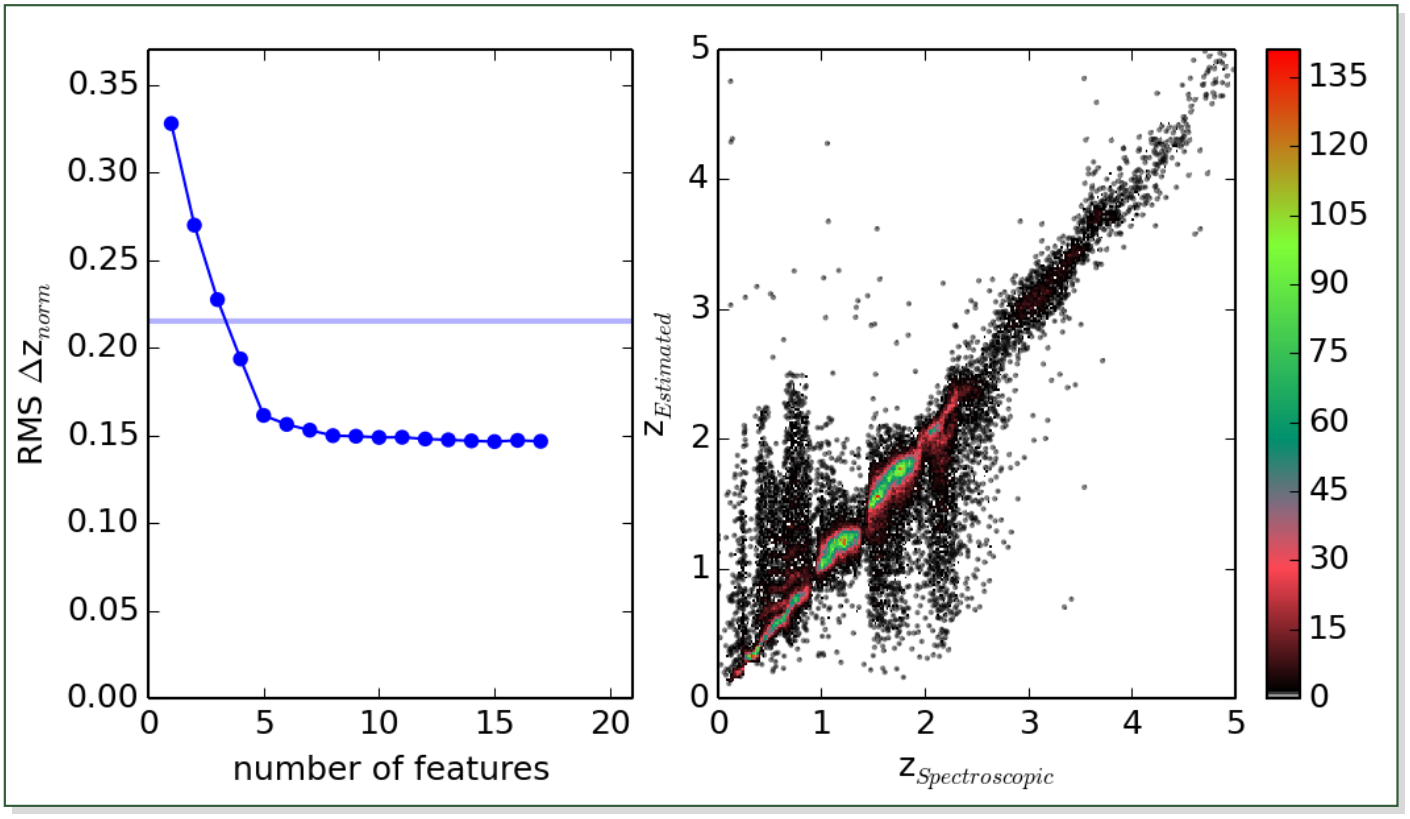
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

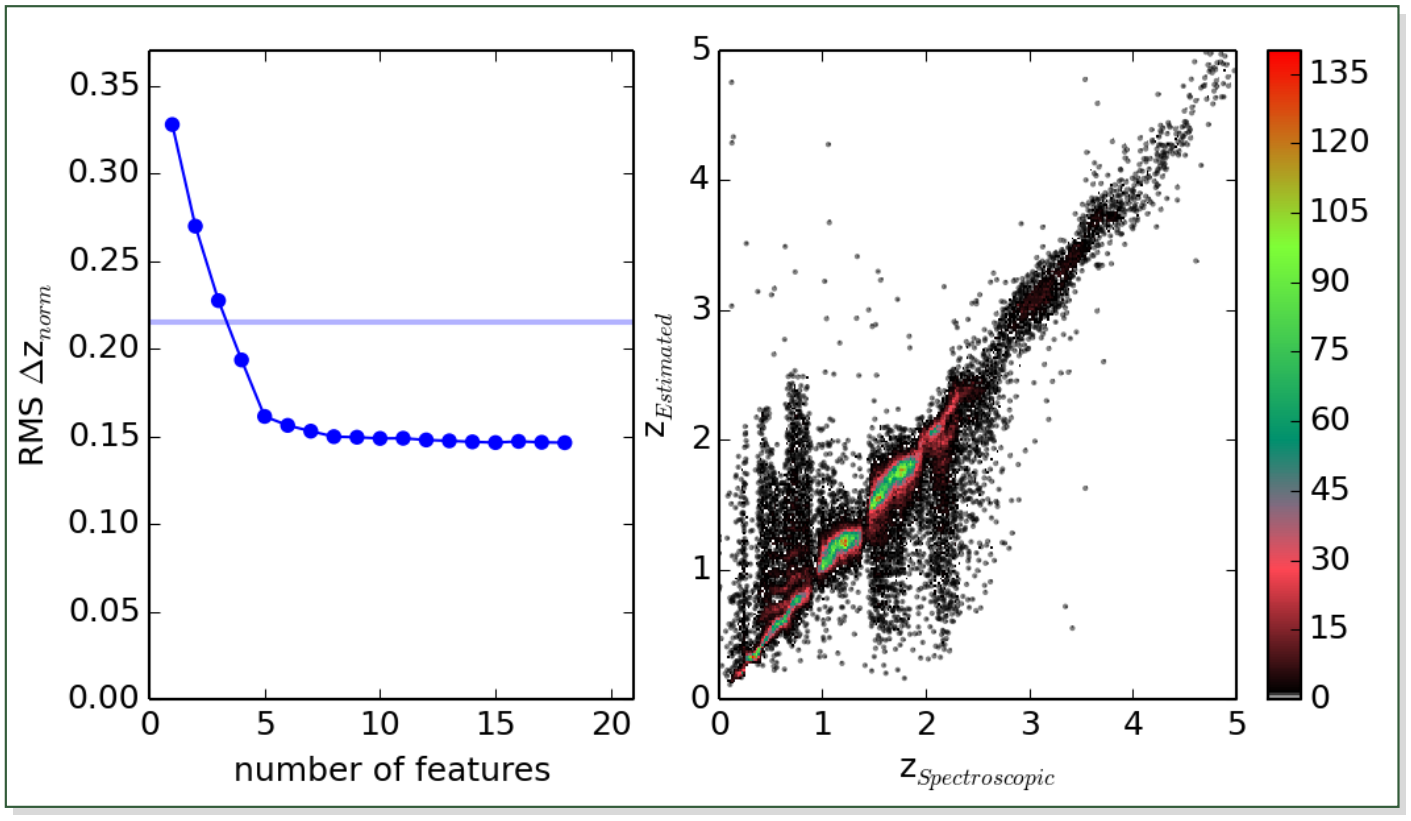
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

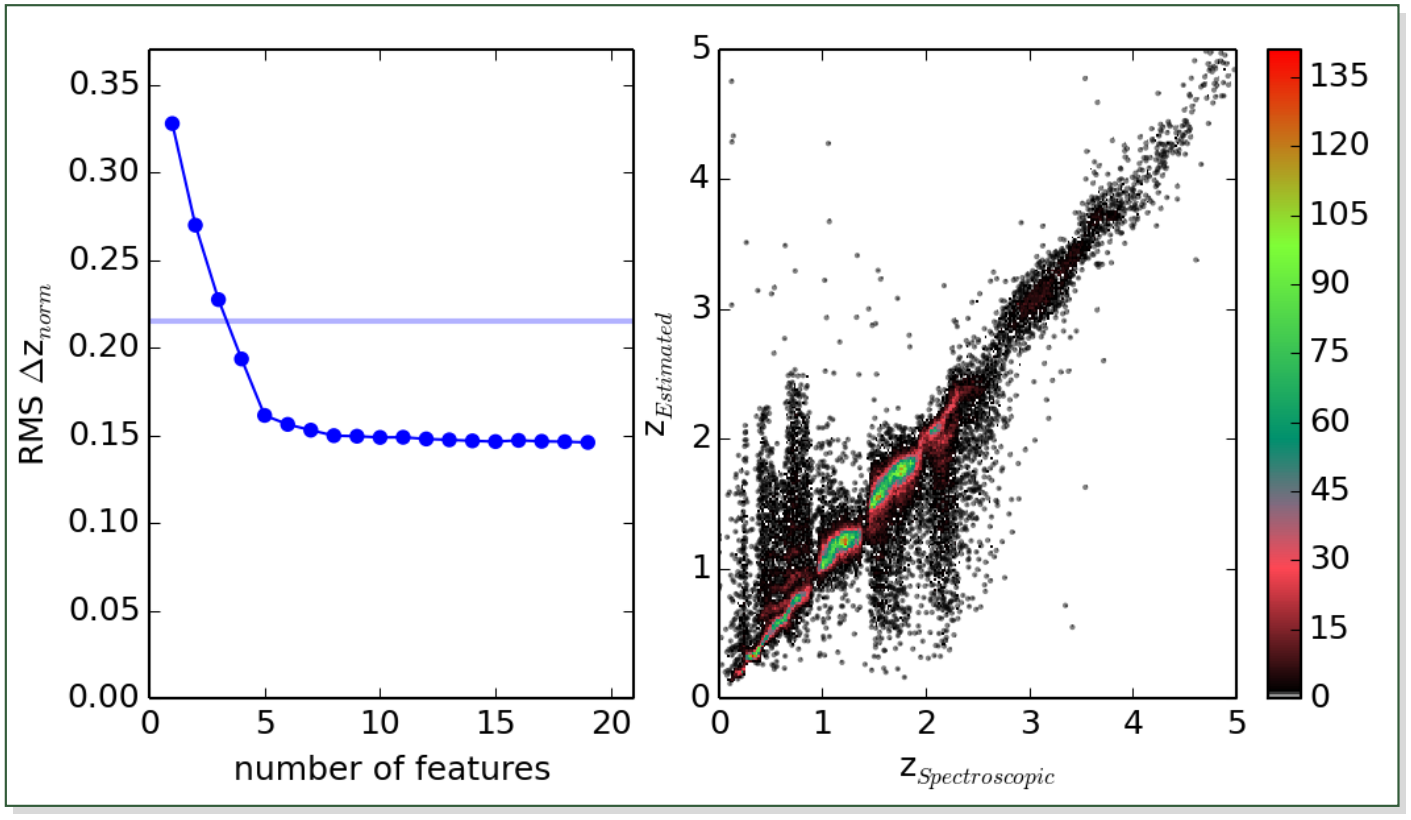
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

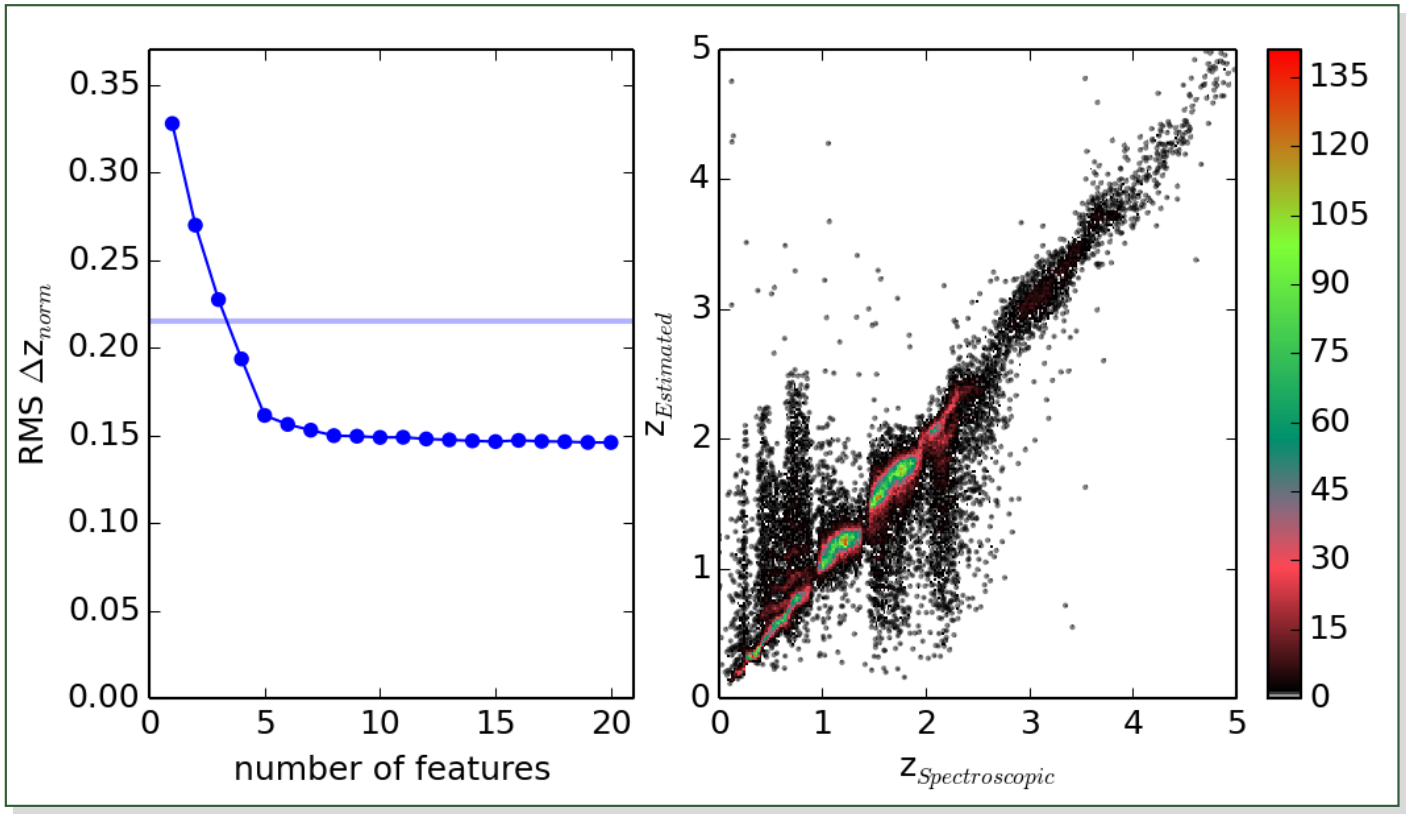
Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

Example: Forward Feature Selection



Forward Selection (Nearest Neighbors)

In each iteration *i*: (1) check all remaining features by computing some validation error (2) add best feature.

Dimensionality Reduction & Approximation

Johnson-Lindenstrauss Lemma

Let $\varepsilon \in (0,1)$ and $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ be an arbitrary set of n points. Further, let k a positive integer such that:

$$k \geq 4 \frac{1}{\frac{1}{2}\varepsilon^2 - \frac{1}{3}\varepsilon^3} \ln n$$

Then, there exists a map $f : \mathbb{R}^d \to \mathbb{R}^k$ such that for all $\mathbf{v}, \mathbf{u} \in S$, we have

$$(1-\varepsilon)\|\mathbf{u}-\mathbf{v}\|^2 \leq \|f(\mathbf{u})-f(\mathbf{v})\|^2 \leq (1+\varepsilon)\|\mathbf{u}-\mathbf{v}\|^2$$

Dasgupta and Gupta. *An Elementary Proof of a Theorem of Johnson and Lindenstrauss*, Random Structures & Algorithms, 2003.

Dimensionality Reduction & Approximation

Johnson-Lindenstrauss Lemma

Let $\varepsilon \in (0,1)$ and $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$ be an arbitrary set of n points. Further, let k a positive integer such that:

$$k \geq 4 \frac{1}{\frac{1}{2}\varepsilon^2 - \frac{1}{3}\varepsilon^3} \ln n$$

Then, there exists a map $f : \mathbb{R}^d \to \mathbb{R}^k$ such that for all $\mathbf{v}, \mathbf{u} \in S$, we have

$$(1-\varepsilon)\|\mathbf{u}-\mathbf{v}\|^2 \leq \|f(\mathbf{u})-f(\mathbf{v})\|^2 \leq (1+\varepsilon)\|\mathbf{u}-\mathbf{v}\|^2$$

$(1+\varepsilon)$-Approximation

- Let $\Phi \in \mathbb{R}^{n \times d}$ a matrix whose entries are sampled i.i.d. from $\mathcal{N}(0,1)$. Consider the mapped data $\mathbf{Z} = \mathbf{X}\Phi \in \mathbb{R}^{n \times k}$.
- This mapping fulfills the conditions with probability of at least $\delta = 1 - \frac{2}{\sqrt{n}}$.
- Do search in low-dimensional space $\mathbb{R}^k$ to obtain a $(1+\varepsilon)$-approximation.

Dasgupta and Gupta. *An Elementary Proof of a Theorem of Johnson and Lindenstrauss*, Random Structures & Algorithms, 2003.

Outline

Summary & Outlook

Today

1. Systems & Computing
2. Spatial Search Structures

Outlook

- Today, 12:15-14:00: Tutorial VirtualBox + Google Colab
- Introduction to TensorFlow (CI)
- Large-Scale Least-Squares & Large-Scale Random Forests (FG)
- Boosted Trees & XGBoost (FG)

References I

J. L Bentley.

Multidimensional binary search trees used for associative searching.
Communications of the ACM, 18(9):509–517, 1975.

Christopher M. Bishop.

Pattern Recognition and Machine Learning.
Springer, 2007.

Sanjoy Dasgupta and Anupam Gupta.

An elementary proof of a theorem of johnson and lindenstrauss.
Random Structures & Algorithms, 22(1):60–65, 2003.

V. Garcia, É. Debreuve, F. Nielsen, and M. Barlaud.

K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching.
In *2010 IEEE International Conference on Image Processing*, pages 3757–3760, 2010.

Isabelle Guyon and André Elisseeff.

An introduction to variable and feature selection.
Journal of Machine Learning Research, 3:1157–1182, 2003.

Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel.

Buffer k-d trees: Processing massive nearest neighbor queries on GPUs.
In *Proceedings of the 31st International Conference on Machine Learning. JMLR W&CP*, volume 32, pages 172–180, 2014.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman.

The Elements of Statistical Learning.
Springer, 2 edition, 2009.

References II

Andrew W. Moore.

An intoductory tutorial on kd-trees.
Technical report, Carnegie Mellon University, 1991.
Extract from Efficient Memory-based Learning for Robot Control (PhD Thesis).
http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.6468.

Stephen M. Omohundrol.

Five balltree construction algorithms.
ftp://ftp.icsi.berkeley.edu/pub/techreports/1989/tr-89-063.pdf, 1989.

P. Ram and A. G. Gray.

Which space partitioning tree to use for search?
In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, pages 656–664, USA, 2013.