The Starving CPU Problem

- or -

How I Learned to Stop Worrying about CPU Speed and Love Memory Access

Francesc Alted
Freelance Trainer And Developer

Advanced Scientific Programming in Python, Split (Croatia)
September 11, 2014

Overview

- Motivation
- The Data Access Issue
  - Why Modern CPUs Are Starving
  - Why Caches?
- Techniques For Fighting Data Starvation
- Optimal Containers for Big Data

Motivation

- or -

Why CPU Speed Is Not

The Holy Grail Any Longer,

but still can help saving the day

(although in an unexpected way!)

The MovieLens Dataset

http://www.grouplens.org/datasets/movielens/

- Datasets for movie ratings
- Different sizes: 100K, 1M, 10M ratings (the 10M will be used in benchmarks ahead)
- The datasets were collected over various periods of time

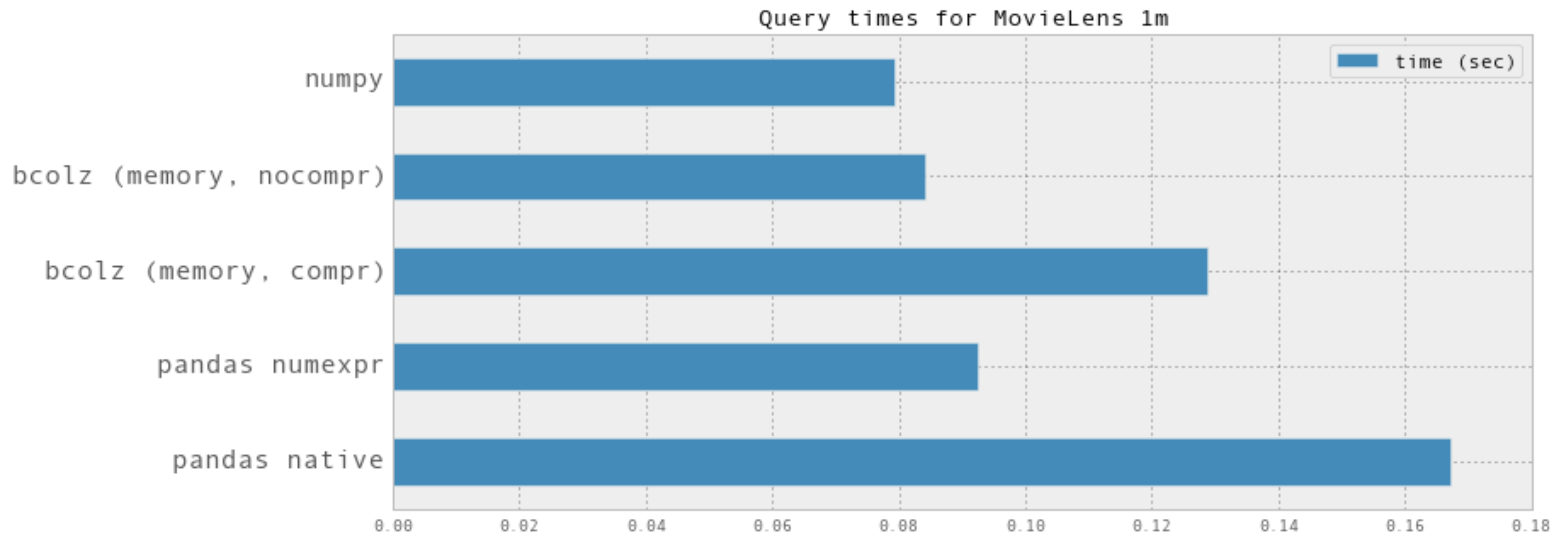
Interactive Session Starts

Materials in:
https://github.com/Blosc/movielens-bench

Look for this IPython notebook:
querying-assp14.ipynb

Based on previous work by Greg Reda:
http://www.gregreda.com/2013/10/26/using-pandas-on-the-movielens-dataset/

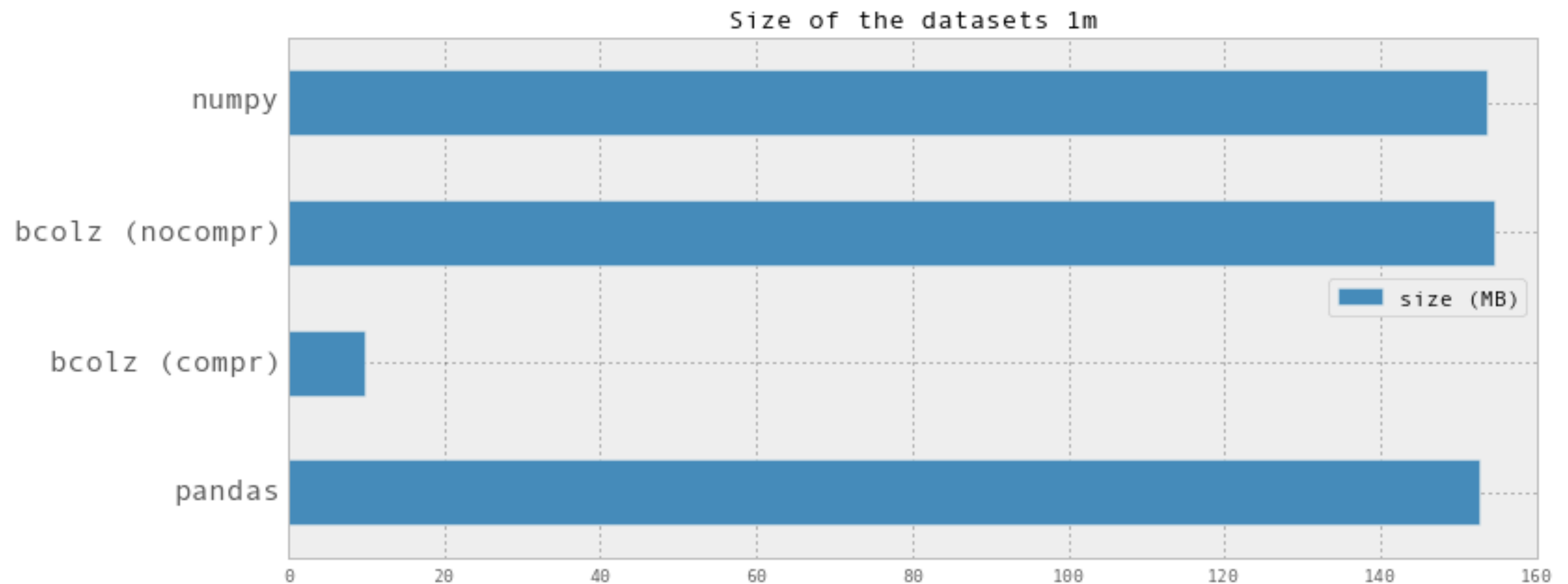
Query Performance (old laptop)



- Compression leads to slightly better query speeds (5% faster)
- Using numexpr really accelerates queries a lot

Note: all computations use a single thread

bcolz Storage Wins (MovieLens Dataset)



- Compressed bcolz allows for 15x win wrt. pandas
- Compression is typically better for larger datasets (exercises)

Query times for MovieLens 1m

time (sec)

numpy

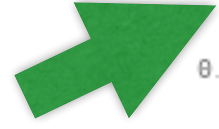
bcolz (memory, nocompr)

bcolz (memory, compr)

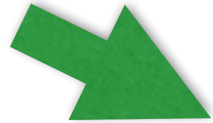
pandas numexpr

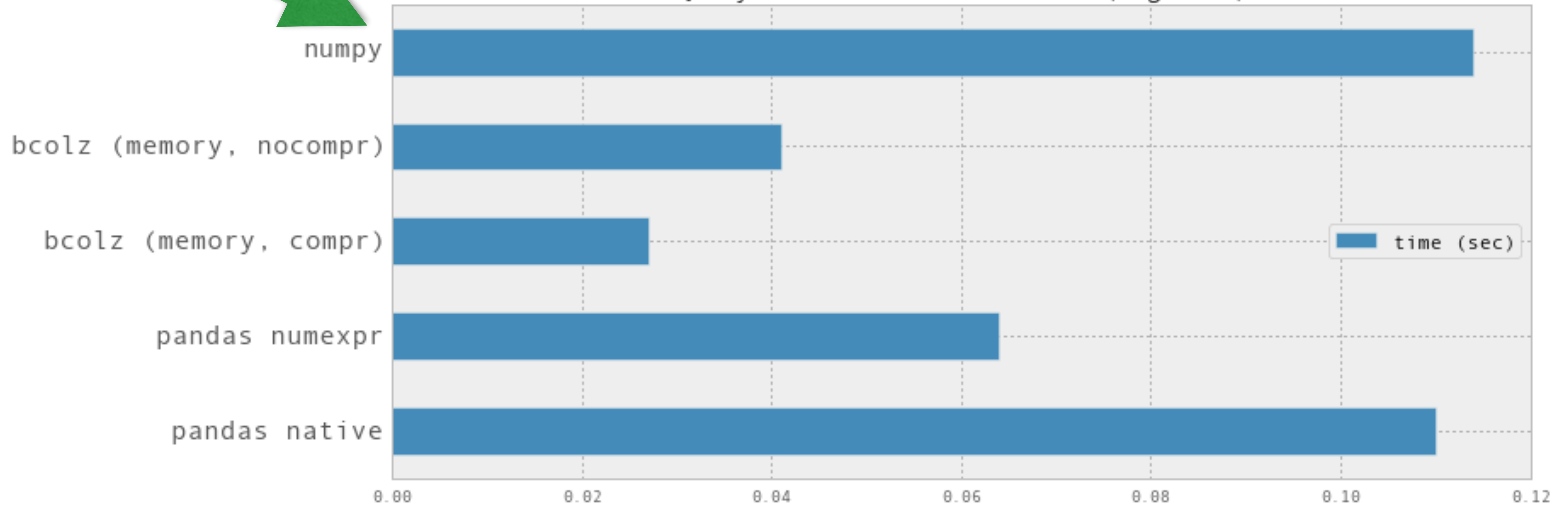
pandas native

0.00 0.02 0.04 0.06 0.08 0.10 0.12 0.14 0.16 0.18

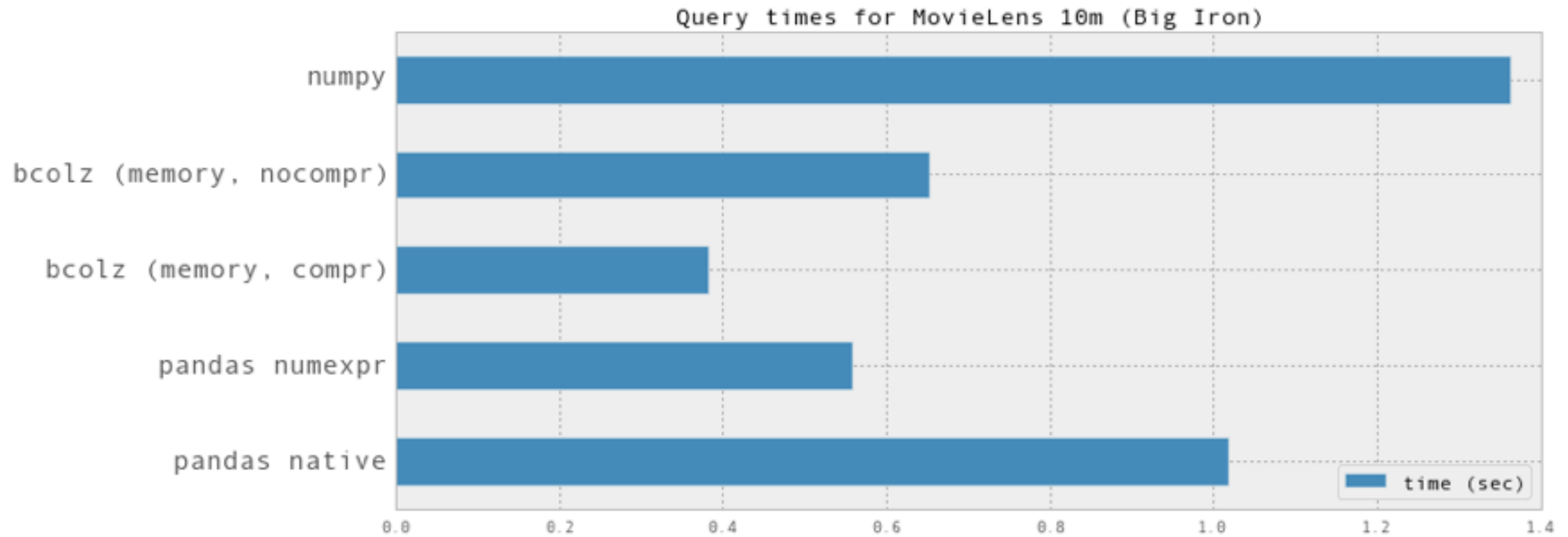


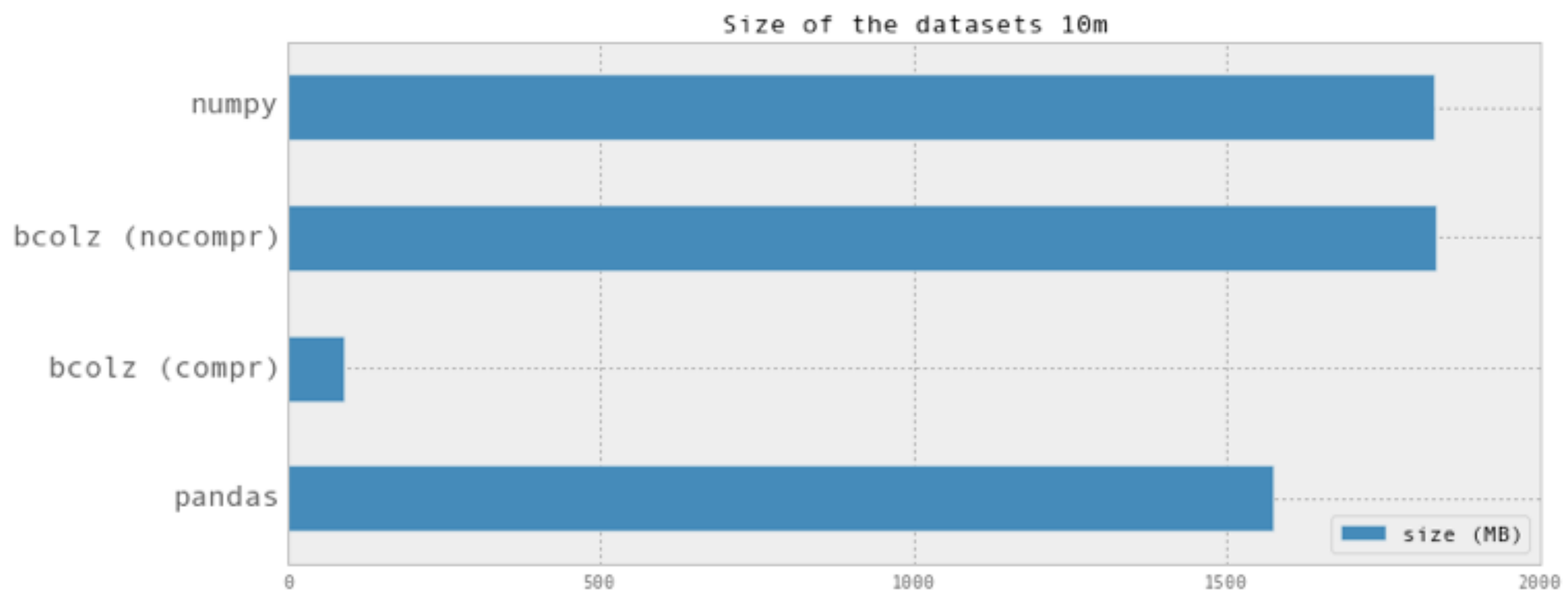
Old laptop
vs
‘Big Iron’





10m records in Big Iron





Some Questions

1. Why numexpr can accelerate computations on pandas, when both packages are provided with computational kernels in C?
2. Why NumPy queries are slow when compared with pandas or bcolz (at least in modern computers)?
3. Why the overhead of compression is so little (and negative in modern computers)?

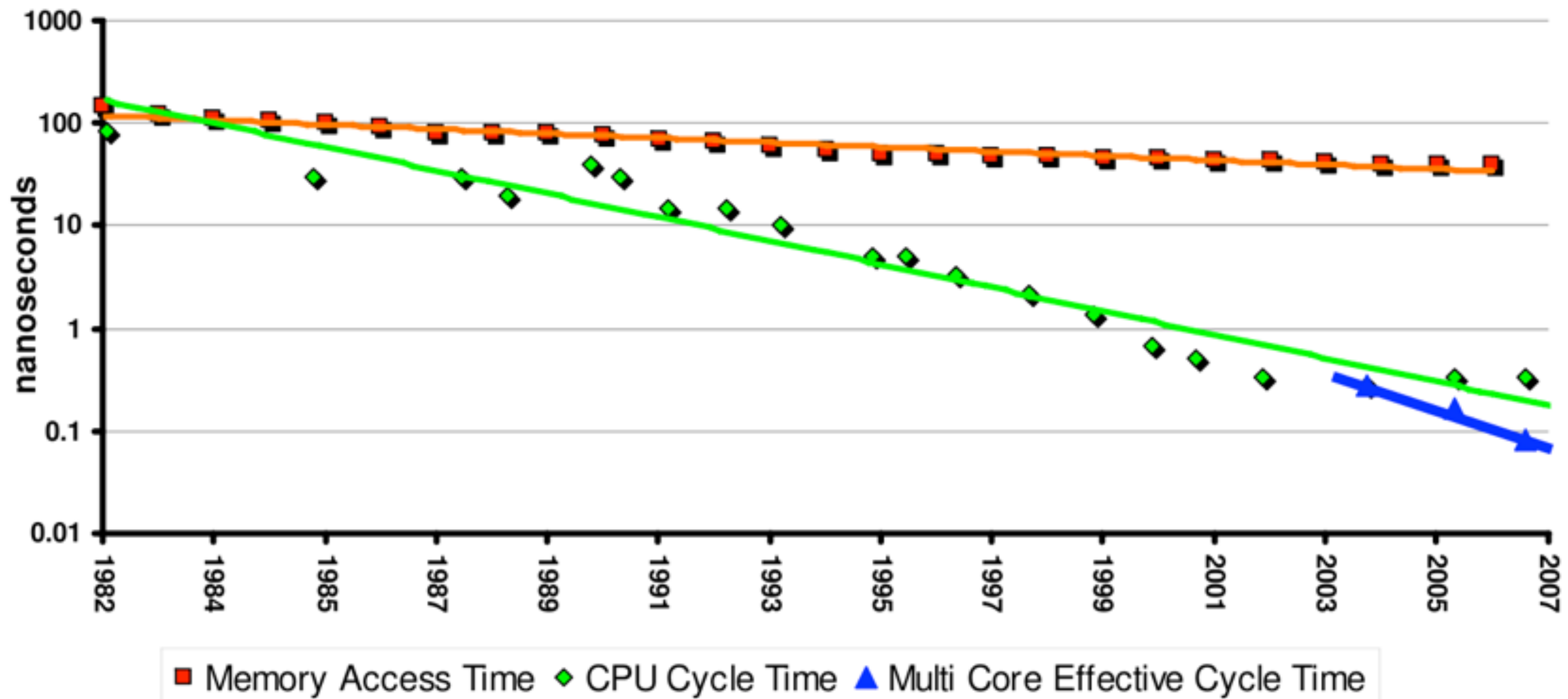
Short answer: **by making a more efficient use of the memory**

The Starving CPU Problem

The Starving CPU Problem

- Current CPUs typically stay bored, doing nothing most of the time
- Why so?
- Because they are basically **waiting for data**

Memory Access Time vs CPU Cycle Time



Quote Back in 1993

“We continue to benefit from tremendous increases in the raw speed of microprocessors without proportional increases in the speed of memory. This means that 'good' performance is becoming more closely tied to good memory access patterns, and careful re-use of operands.”

“No one could afford a memory system fast enough to satisfy every (memory) reference immediately, so vendors depends on caches, interleaving, and other devices to deliver reasonable memory performance.”

– Kevin Dowd, after his book “High Performance Computing”, O’Reilly & Associates, Inc, 1993

Quote Back in 1996

"Across the industry, today's chips are largely able to execute code faster than we can feed them with instructions and data. There are no longer performance bottlenecks in the floating-point multiplier or in having only a single integer unit. The real design action is in memory subsystems— caches, buses, bandwidth, and latency."

"Over the coming decade, memory subsystem design will be the only important design issue for microprocessors.."

– Richard Sites, after his article "It's The Memory, Stupid!", Microprocessor Report, 10(10),1996

Book in 2009

Copyrighted Material

MORGAN & CLAYPOOL PUBLISHERS

The Memory System

You Can't Avoid It,
You Can't Ignore It,
You Can't Fake It

Bruce Jacob

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Mark D. Hill, *Series Editor*

Copyrighted Material

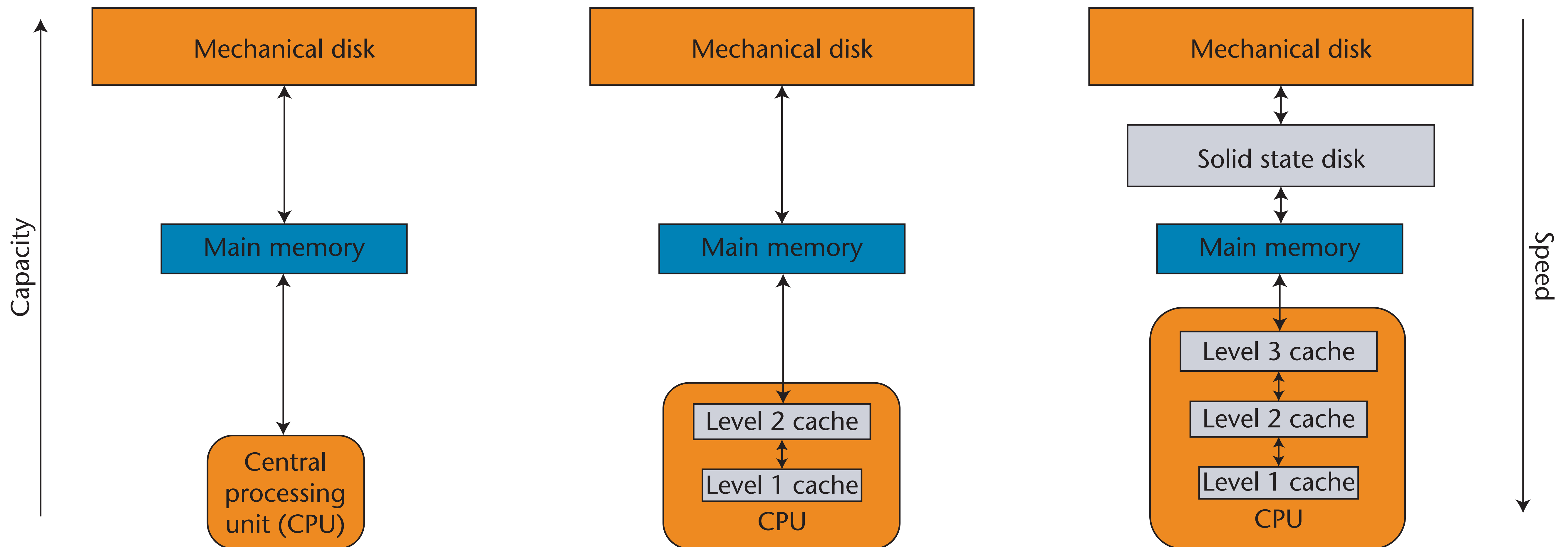
The Status of CPU Starvation in 2014

- Memory latency is much slower (between 100x and 250x) than processors.
- Memory bandwidth is improving at a better rate than memory latency, but it is also lagging behind processors (between 30x and 100x).

CPU Caches to the Rescue

- Vendors realized the Starving CPU problem and responded implementing caches in CPUs
- CPU cache latency and throughput are much better than memory
- However: the faster they run the smaller they must be

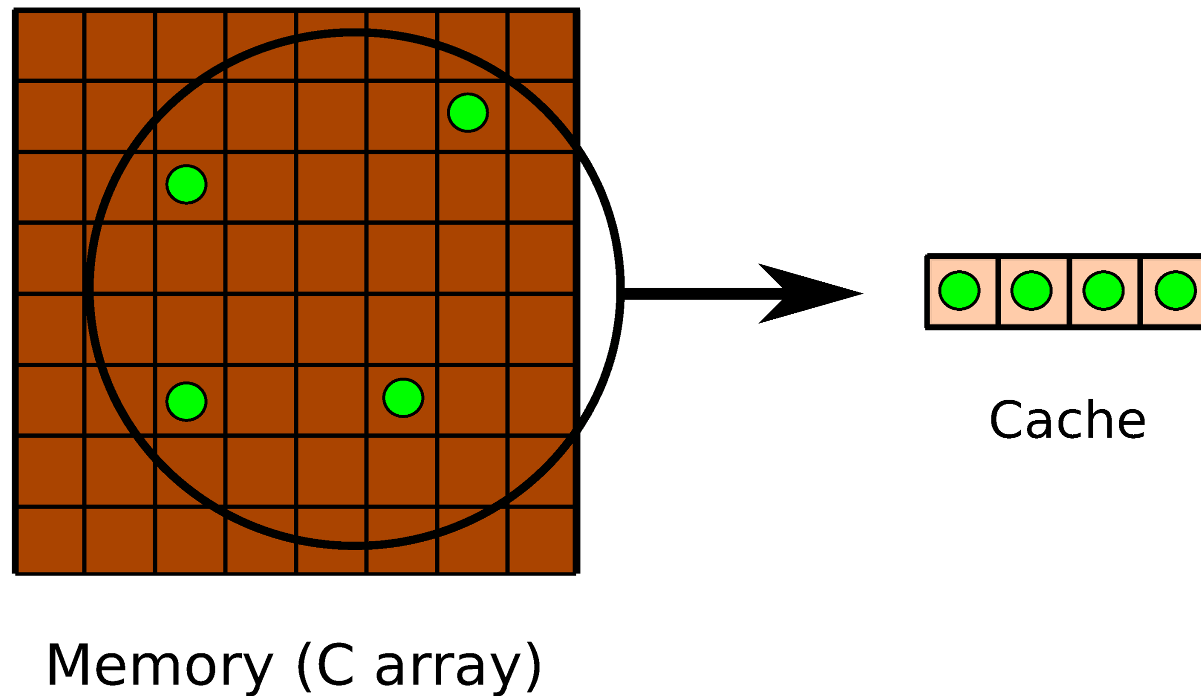
CPU Cache Evolution



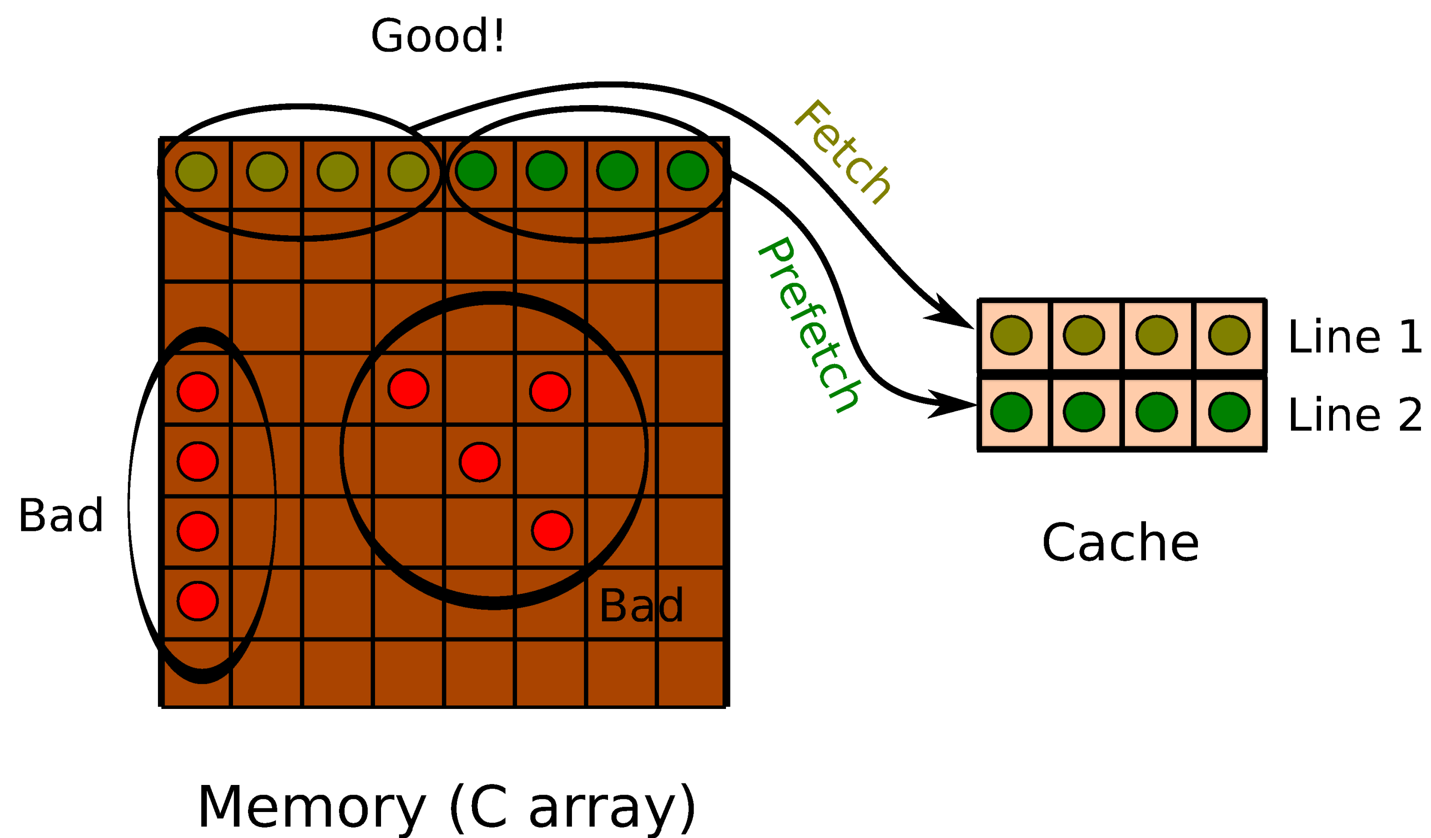
When CPU Caches Are Effective?

- Mainly in a couple of scenarios:
  - Time locality: when the dataset is reused
  - Spatial locality: when the dataset is accessed sequentially

Parts of the dataset are reused

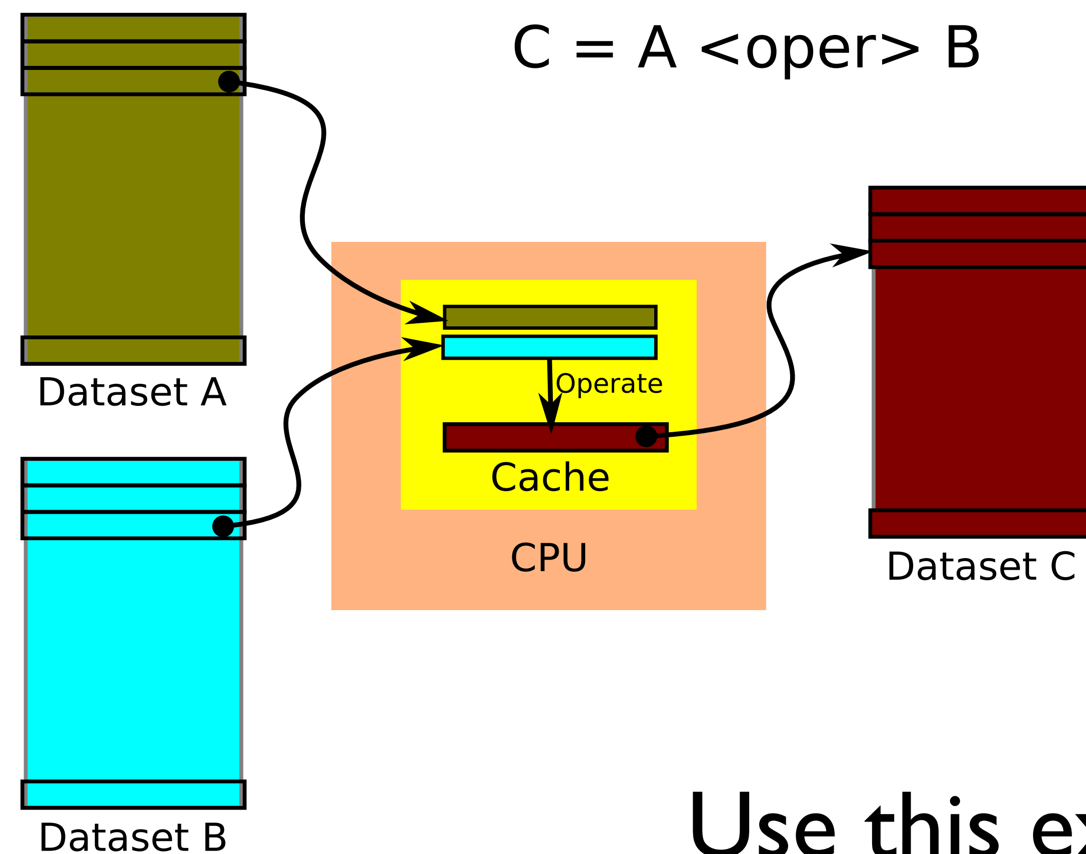


Dataset is accessed sequentially



ue

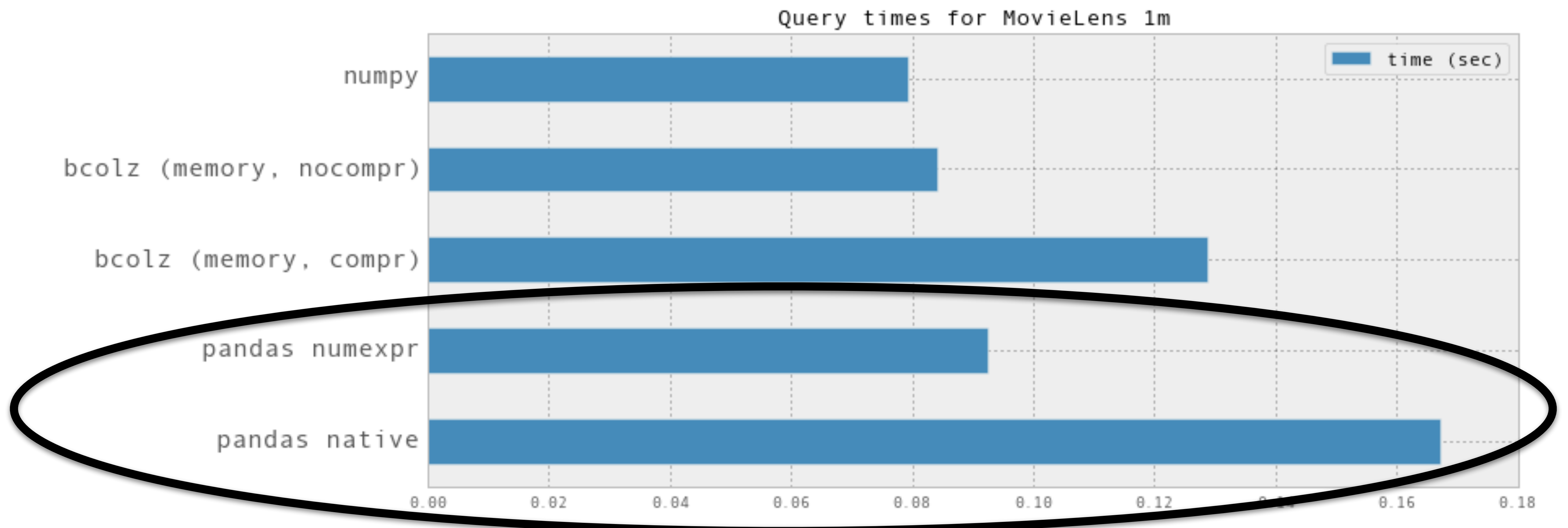
When accessing disk or memory, get a contiguous block that fits in CPU cache, operate upon it and reuse it as much as possible.



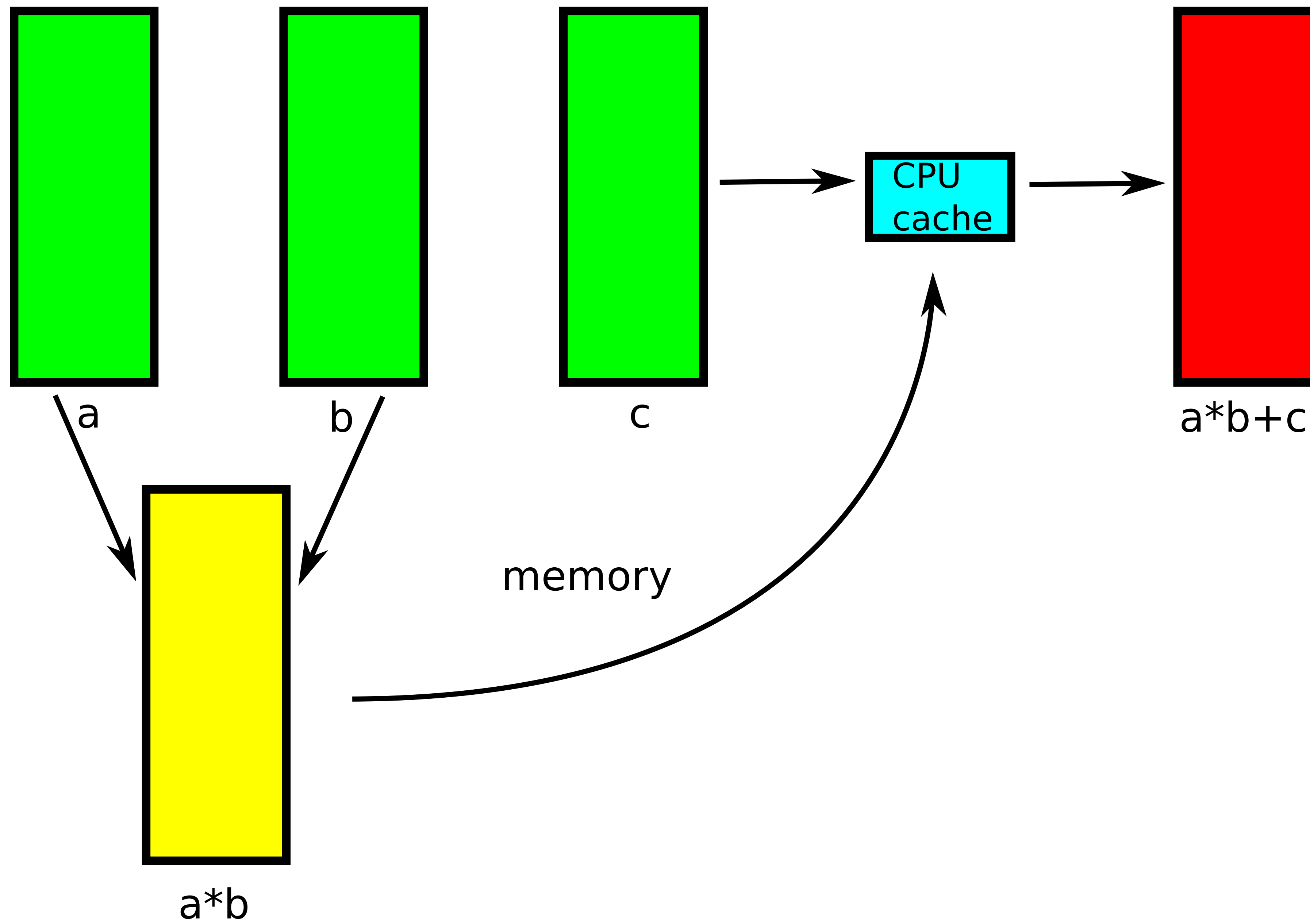
Use this extensively to leverage spatial and temporal localities

Time To Answer Pending Question #1

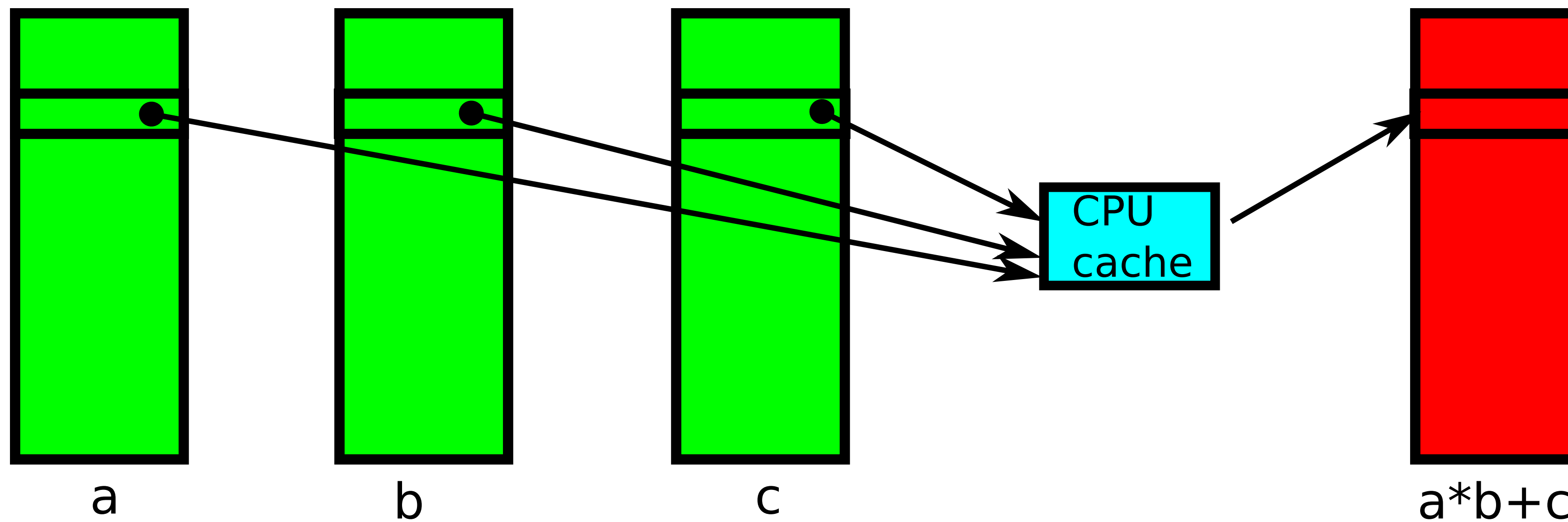
```
lens.query("(title == 'Tom and Huck (1995)') & (rating == 5)”)
vs
lens[(lens['title'] == 'Tom and Huck (1995)') & (lens['rating'] == 5)]
```

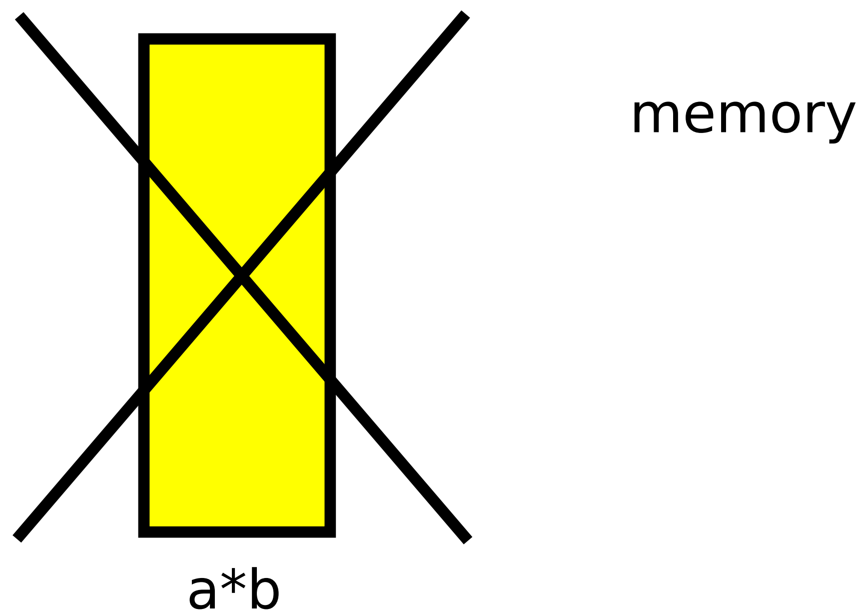


Computing "a*b+c" with NumPy. Temporaries goes to memory.



Computing "a*b+c" with Numexpr. Temporaries in memory are avoided.



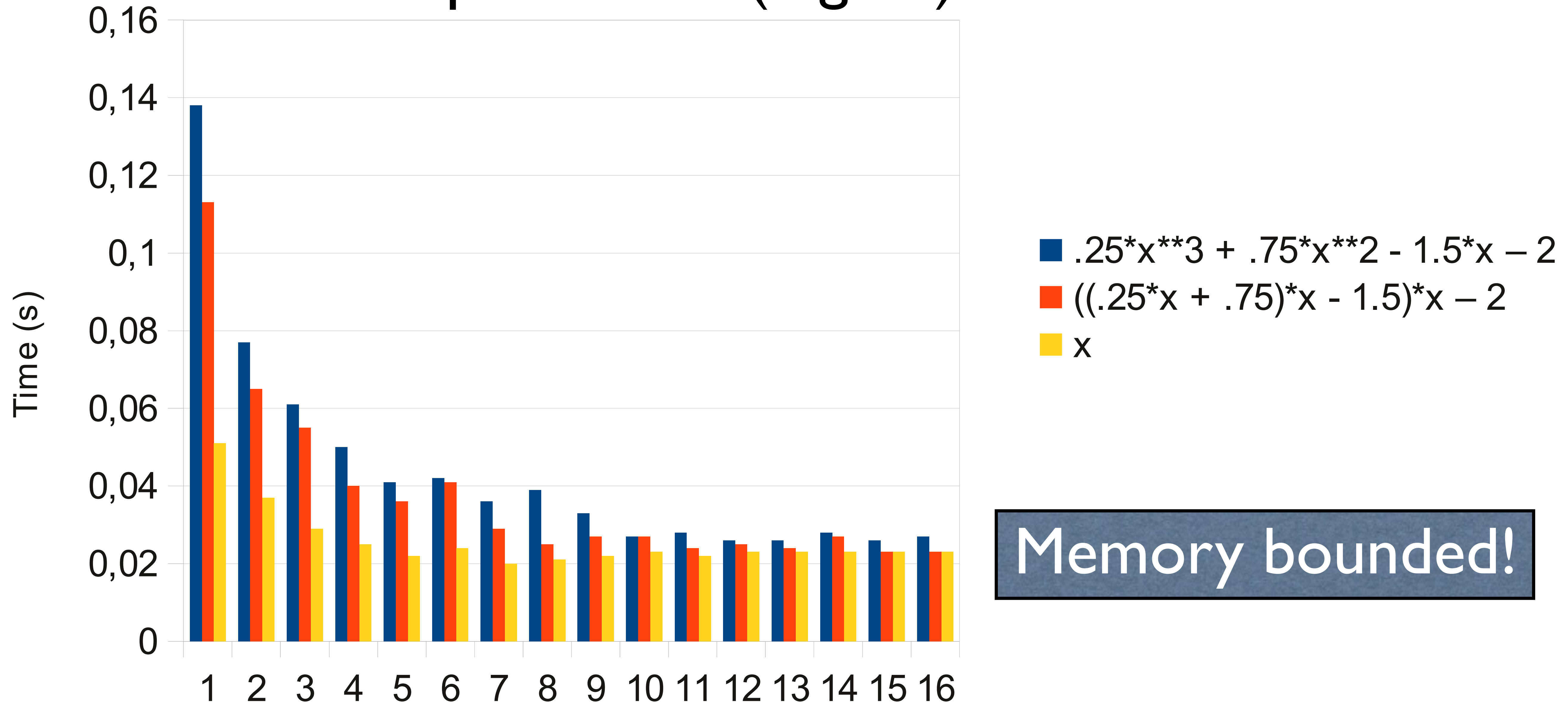


memory

Blocking technique at work!

Multithreaded numexpr

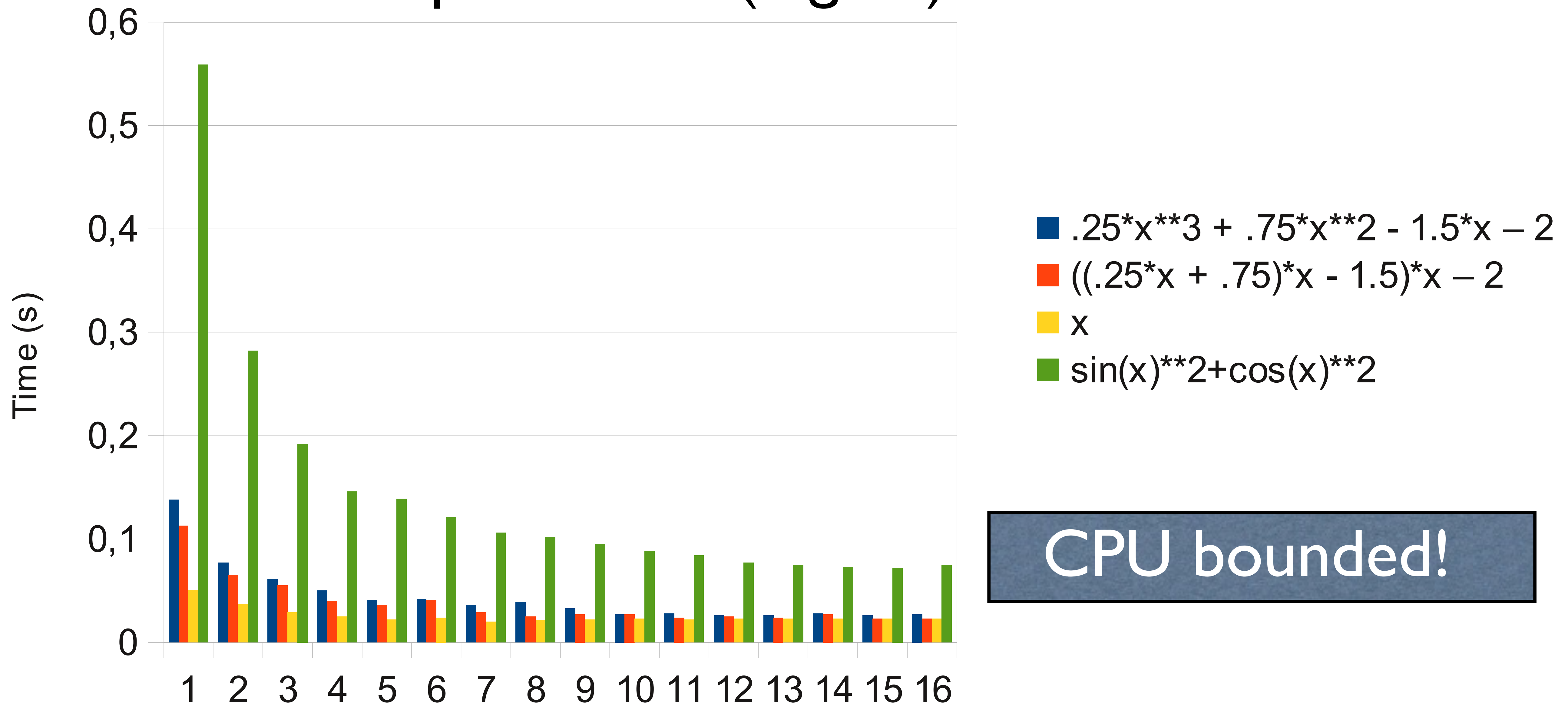
numexpr with 16 (logical) cores



((.25*x + .75)*x - 1.5)*x – 2

Transcendental Functions

numexpr with 16 (logical) cores



Numexpr Limitations

- Numexpr only implements element-wise operations, i.e. ‘*a*b*’ is evaluated as:

```
for i in range(N):
    c[i] = a[i] * b[i]
```

- In particular, it cannot deal with things like:

```
for i in range(N):
    c[i] = a[i-1] + a[i] * b[i]
```

First Take Away Message

- Before spending too much time optimizing by yourself make you a favor:

  Use the existing, powerful libraries out there

- It is pretty difficult to beat performance professionals!

Optimal Containers for Big Data

The Need for a Good Data Container

- Too many times we are focused on computing as fast as possible
- But we have seen how important data access is
- Hence, having an optimal data structure is critical for getting good performance when processing very large datasets

No Silver Bullet

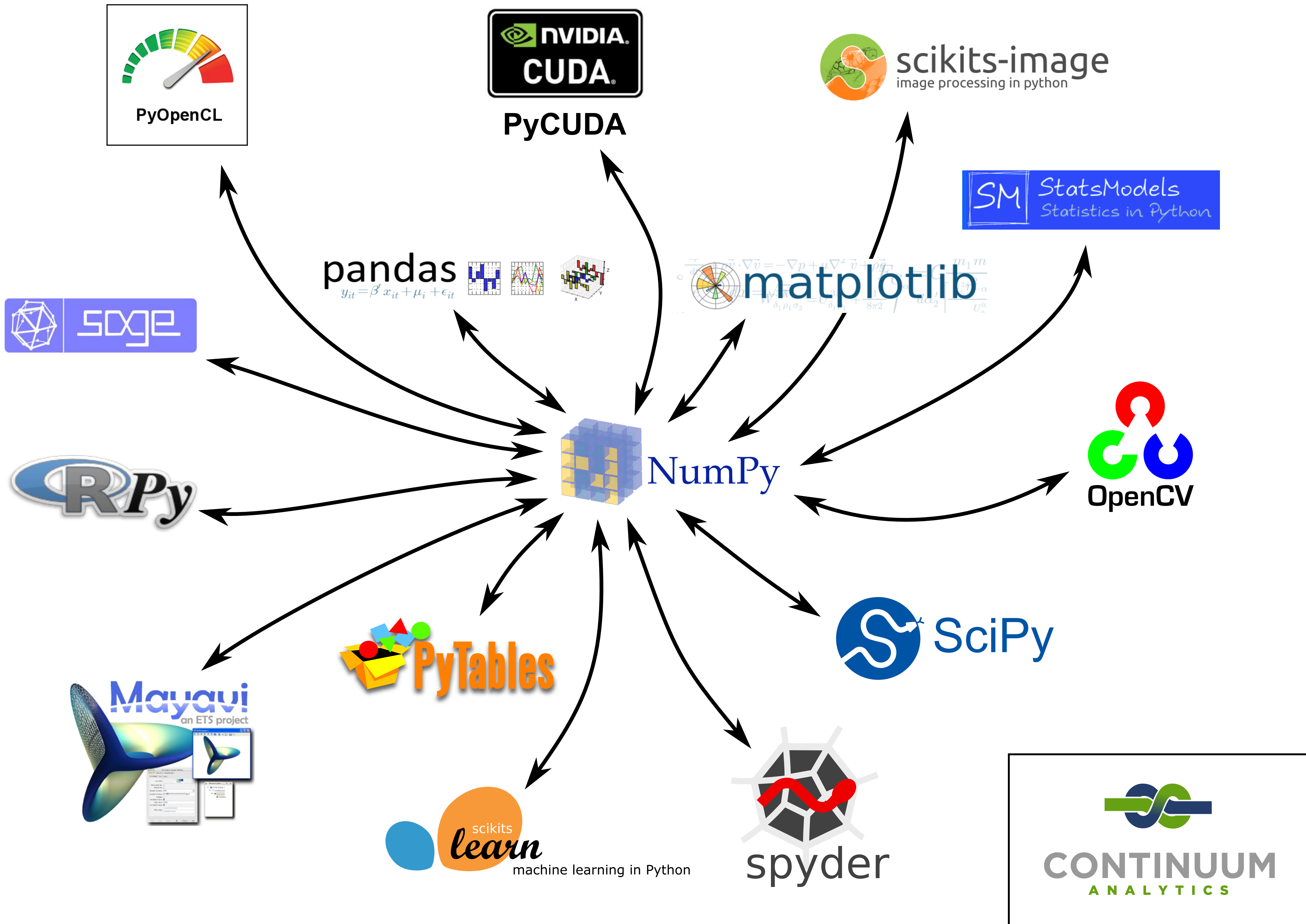
- Unfortunately, there is not (and probably will never be) a fit-all data container
- We need to make our mind to the fact that we need to choose the ‘optimal’ container for **our case**

Plenty of Data Containers Out There

- In-Memory:
  - Python: list, tuple, dict, set, heap, queue…
  - NumPy: multidim arrays, structured arrays
  - pandas: Series, Dataframe, Panel
- Out-of-memory: RDBMs, HDF5, NetCDF and a huge lot more!

NumPy: A De Facto Data Container

NumPy is the standard de facto in-memory container for Big Data applications in the Python universe


PyOpenCL
NVIDIA
CUDA
PyCUDA
scikits-image
image processing in python
SM StatsModels
Statistics in Python
pandas
matplotlib
sage
NumPy
OpenCV
RPy
PyTables
SciPy
Mayavi
an ETS project
scikits
learn
machine learning in Python
spyder
CONTINUUM
ANALYTICS

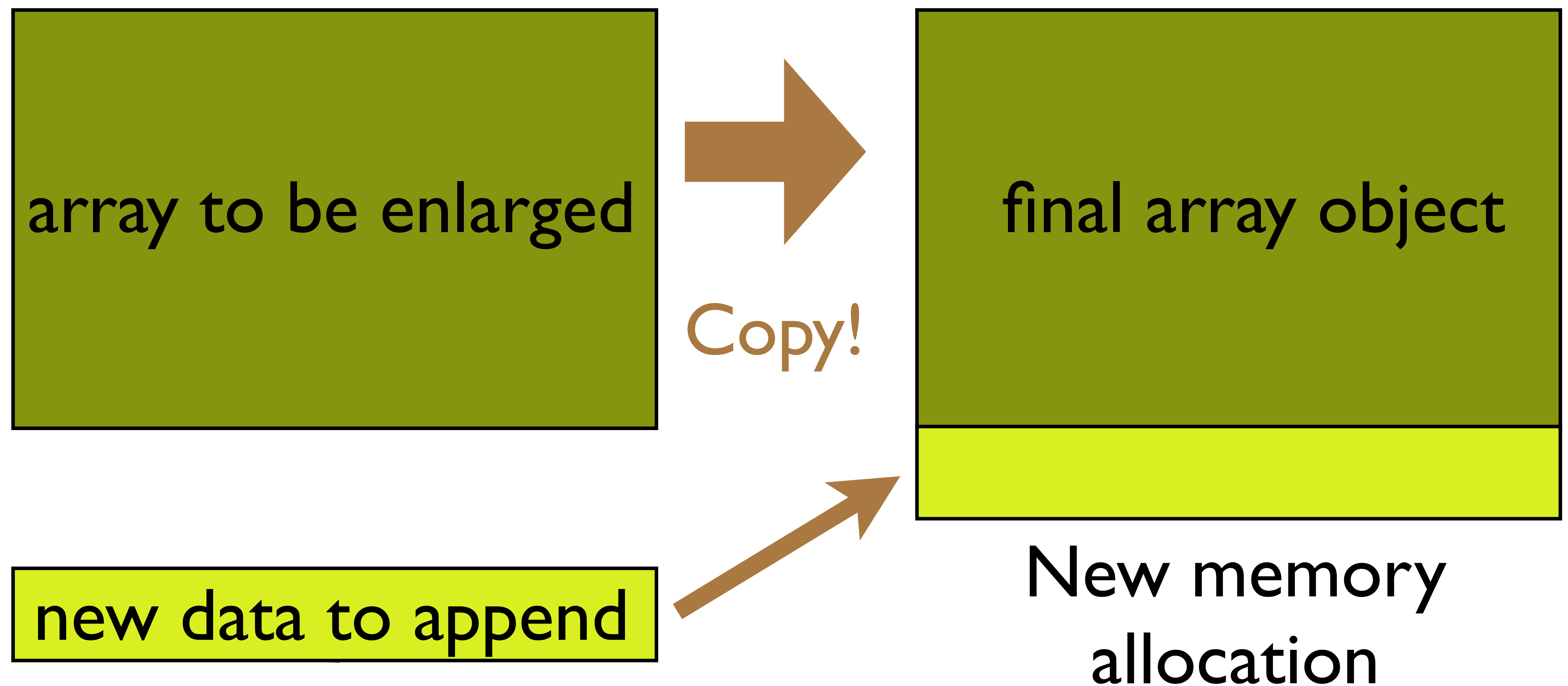
NumPy Advantages

- Multidimensional data container
- Efficient data access for many scenarios
- Powerful weaponry for data handling
- Efficient in-memory storage

Nothing Is Perfect

- The NumPy container is just great for many use cases
- However, it also has its own deficiencies:
  - Not efficient for appending data (so data containers tend to be **static**)
  - Cannot deal with compressed data transparently
  - Limited disk-based data support

Appending Data in Large NumPy Objects



- Normally a `realloc()` syscall will not succeed
- Both memory areas have to exist **simultaneously**

pandas & bcolz

Overcoming some NumPy Limitations

pandas

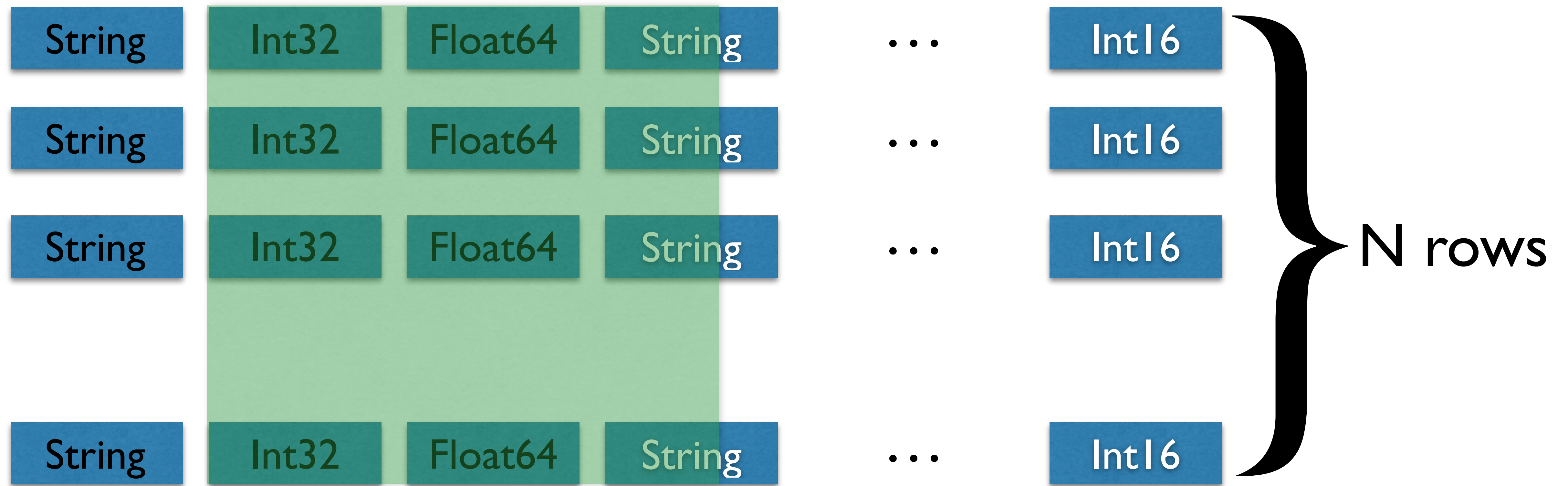
- An easy-to-use library for handling data **very efficiently**
- Data containers: Series, DataFrame, Panel
  - **Dataframe** is **columnar** storage for **tabular** data and the most used one
- Fast interface for persistent formats: CSV, RDBMs, HDF5 and many more!

Why Columnar?

- When querying tabular data, only the **interesting** data is accessed
- Less I/O required

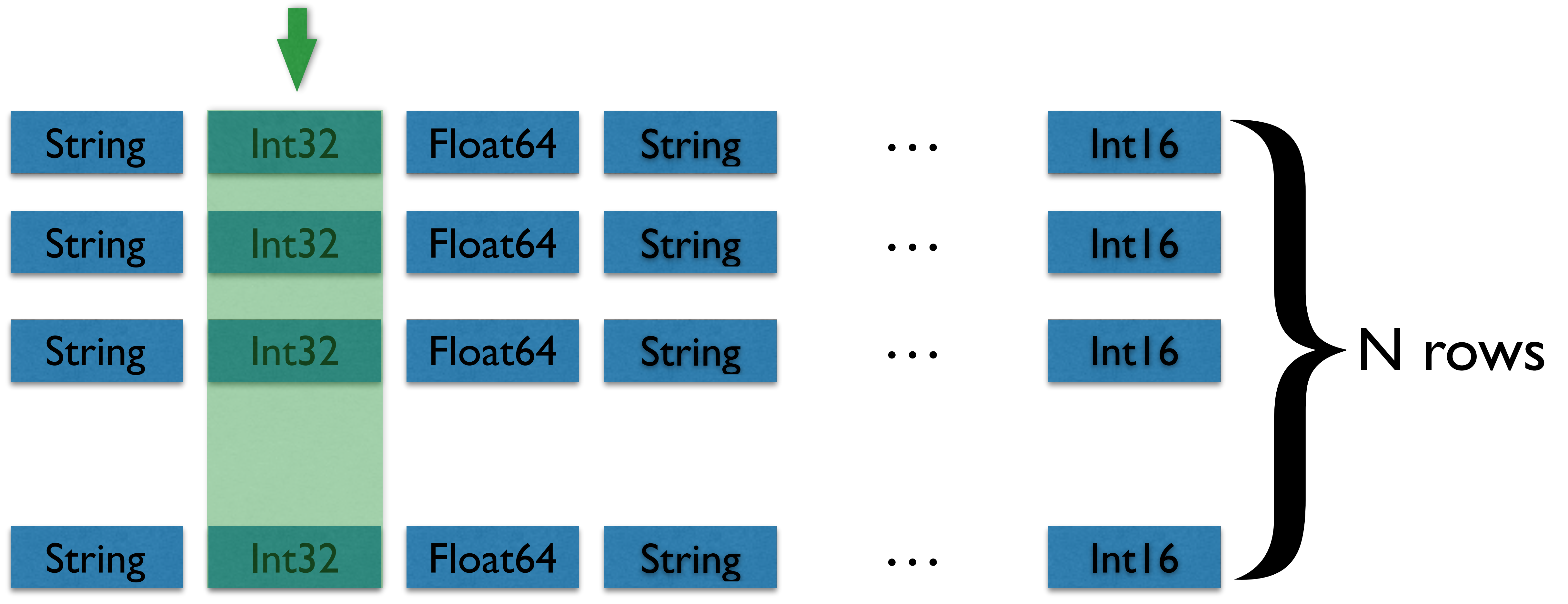
In-Memory Row-Wise Table (Structured NumPy array)

Interesting column



Interesting Data: N * 4 bytes (Int32)
Actual Data Read: N * 64 bytes (cache line)

In-Memory Column-Wise Table (pandas, bcolz)



Interesting Data: N * 4 bytes (Int32)
Actual Data Read: N * 4 bytes (Int32)

Time To Answer Pending Question #2

Why NumPy queries are slow when compared with pandas or bcolz (at least in modern computers)?

Because NumPy structured arrays are row-wise

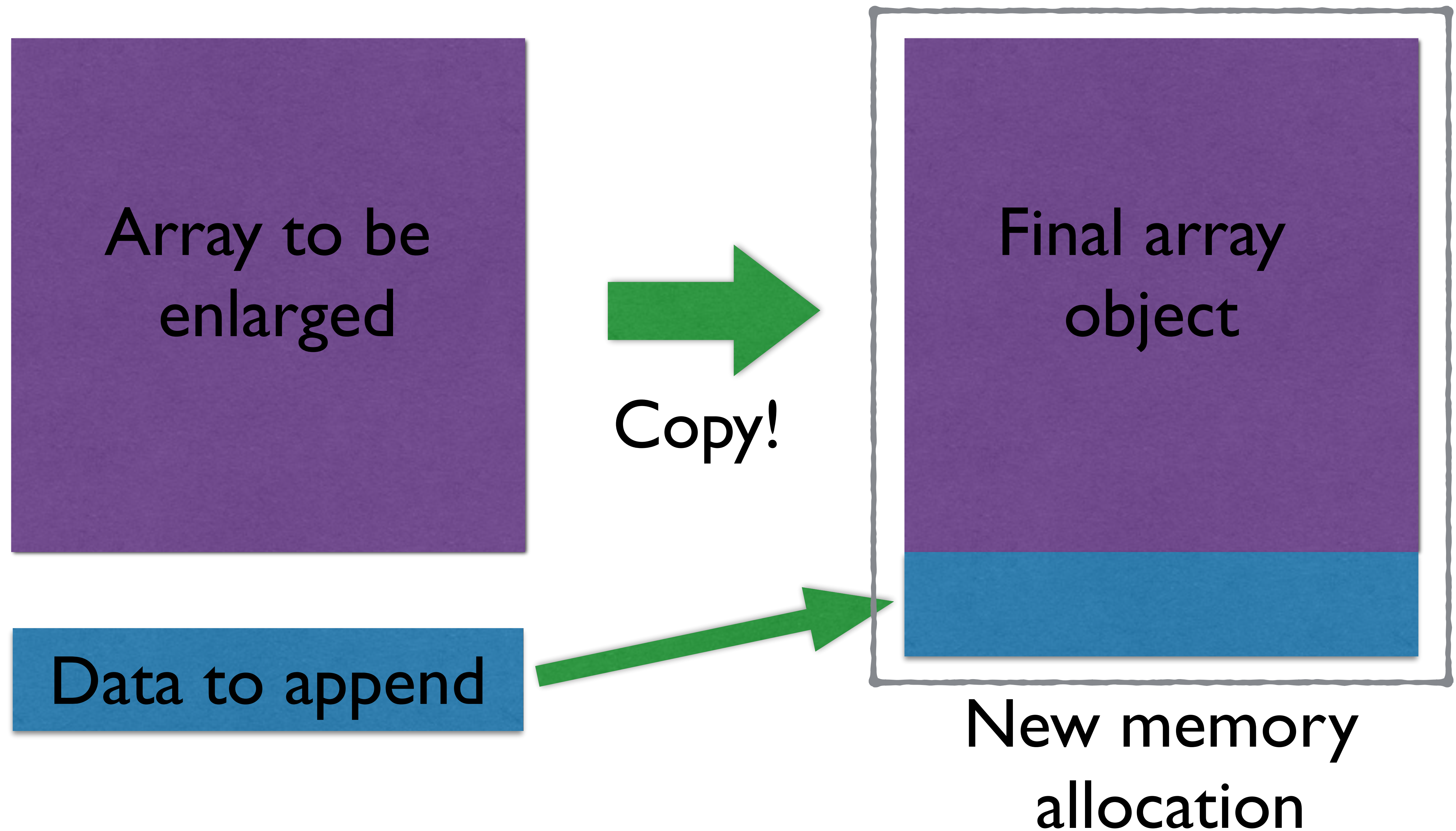
bcolz

- **Columnar, chunked, compressed** data containers for Python
- Offers **carray** (homogeneous) and **ctable** (tabular) container flavors
- Implements **fast iterators** over the containers, **supporting query semantics**
- Uses the fast **Blosc** compressor under the hood

Why Chunking?

- Chunking means more difficulty handling data, so why bother?
  - Efficient enlarging and shrinking
  - On-flight compression possible

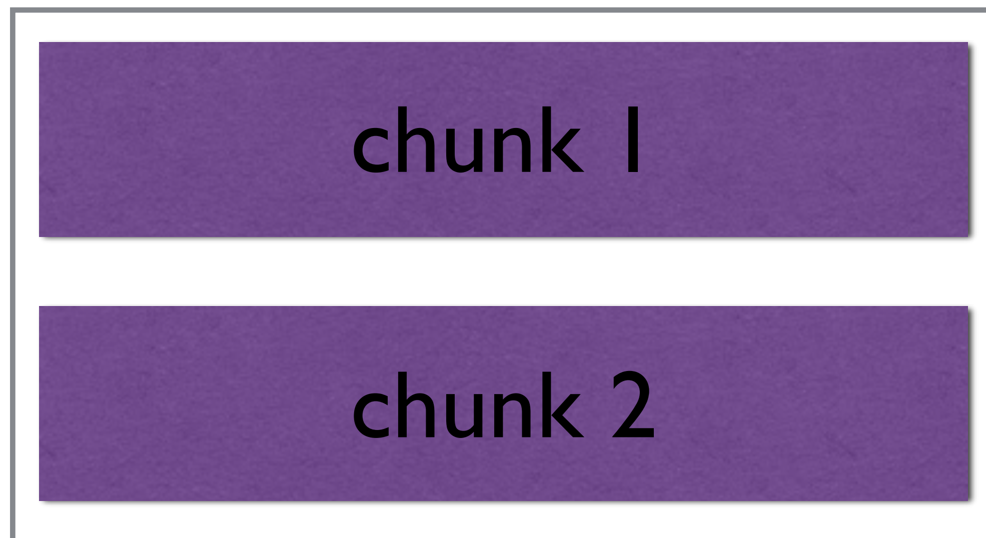
Appending Data in NumPy

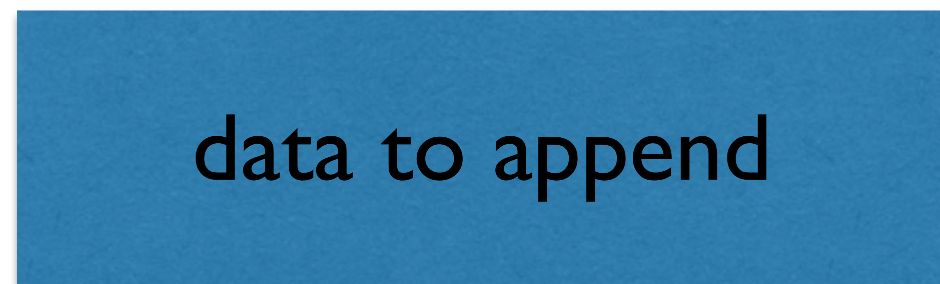


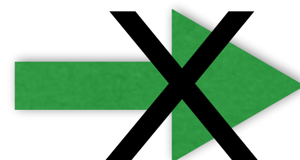
- Both memory areas have to exist **simultaneously**

Appending Data in bcolz

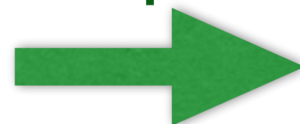
carray to be enlarged



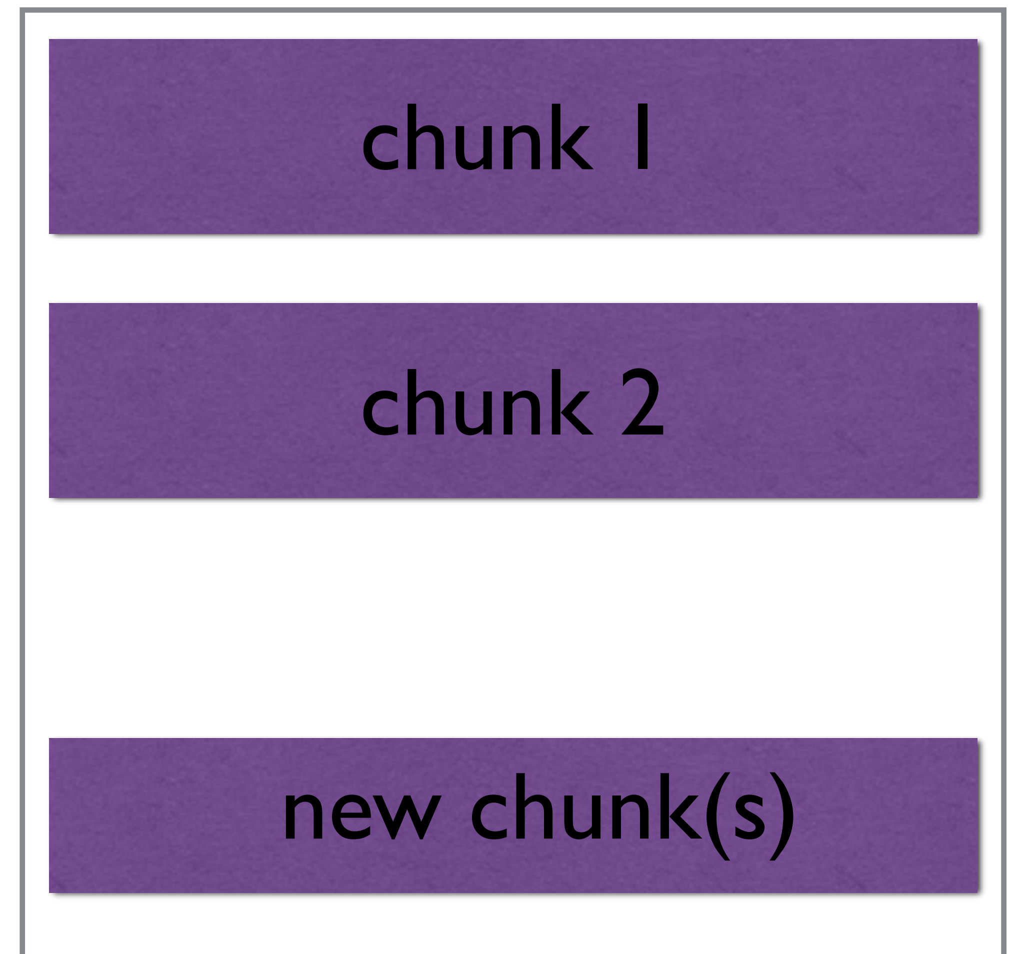




Compress



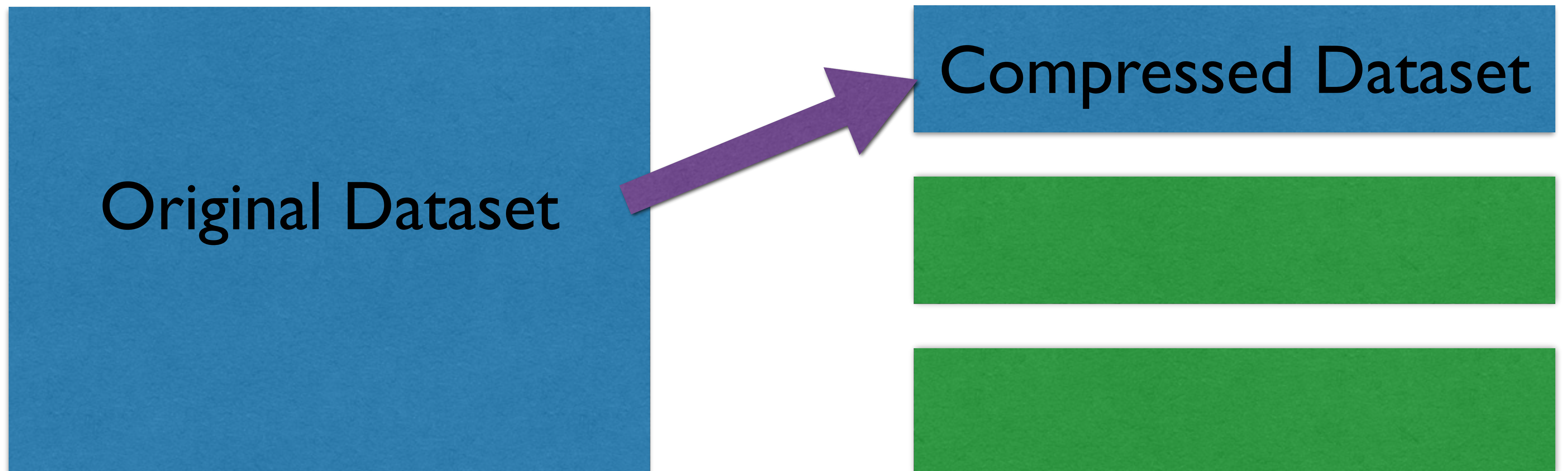
Final carray object



- Only a compression operation on new data is required

Why Compression (I)?

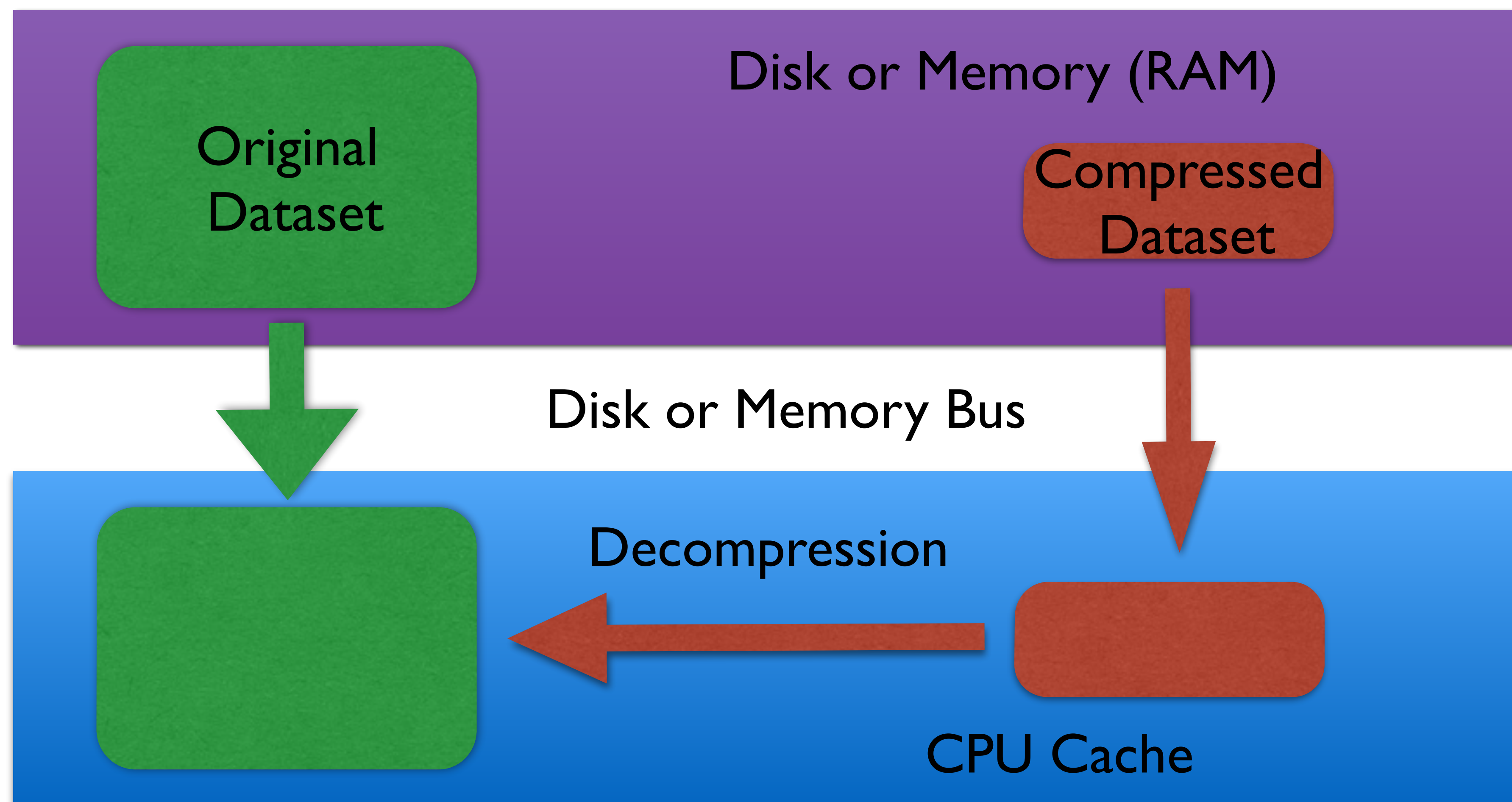
More data can be stored in the same amount of media



3x more data

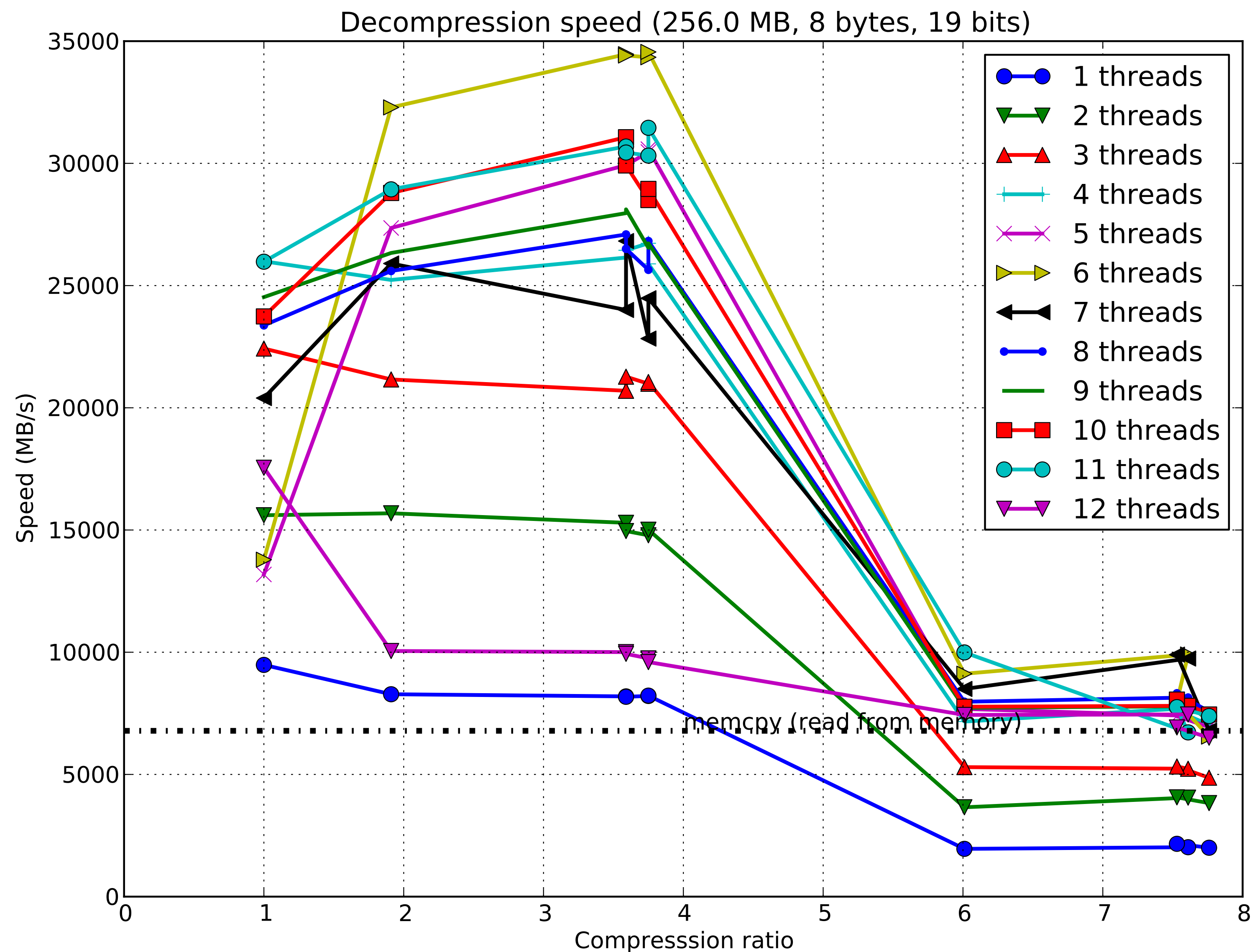
Why Compression (II)?

Less data needs to be transmitted to the CPU

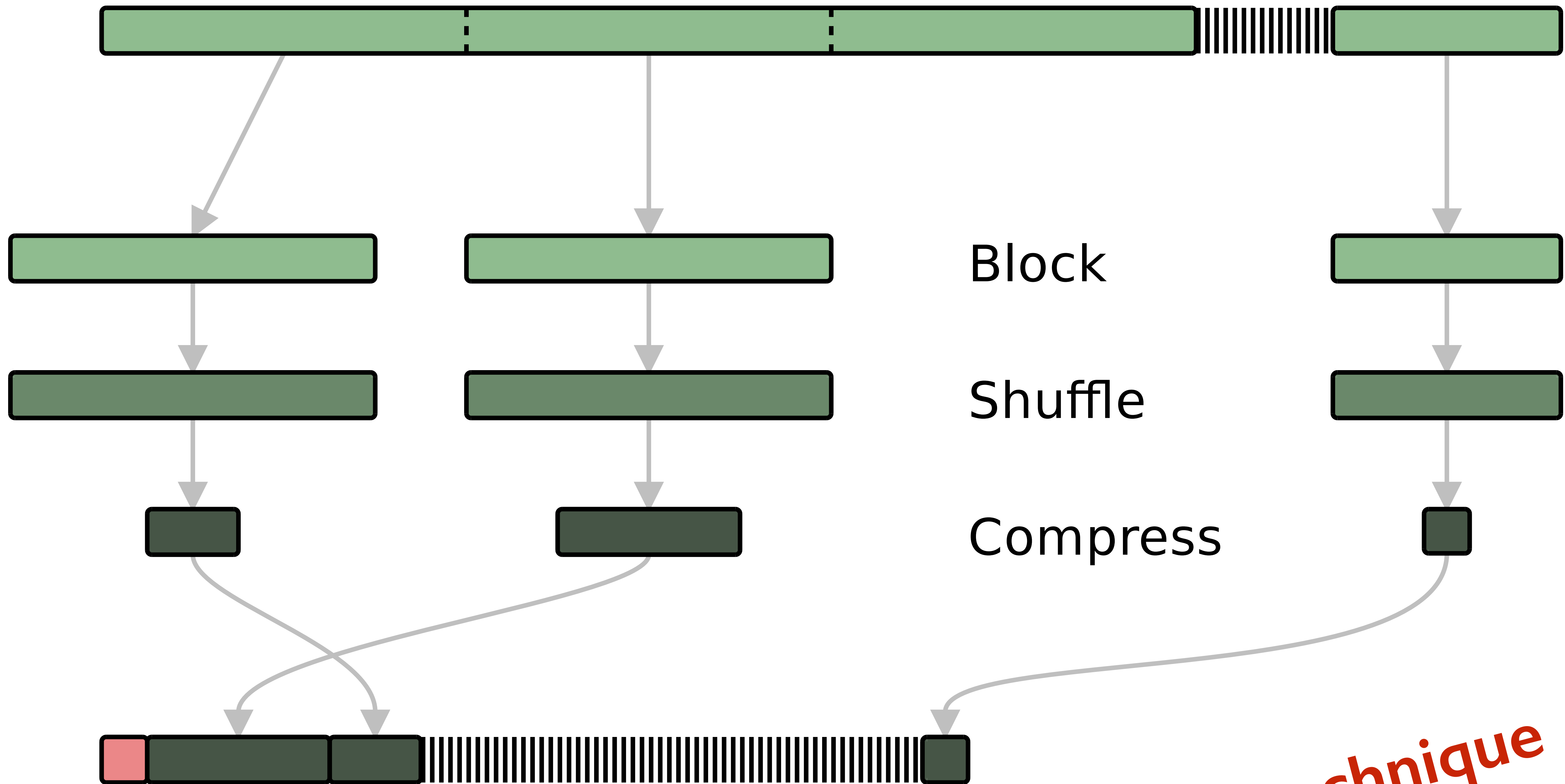


Transmission + decompression faster than direct transfer?

Blosc: Compressing Faster Than Memory Speed



How Blosc Works



Attr: Valentin Haenel

Time To Answer Pending Question #3

Why the overhead of compression is so little (if any)?

Because Blosc can (de)compress faster than memcpy

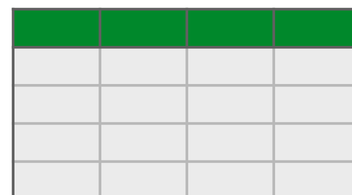
Streaming Analytics with **bcolz**

map(), filter(),
groupby(), sortby(),
reduceby(),
join()

itertools,
PyToolz,
CyToolz

iter(), iterblocks(),
where(), whereblocks(),
__getitem__()

bcolz
iterators/filters
with blocking



bcolz container
(disk or memory)

Final Take Away Message for Today

- Large datasets are tricky to manage, so:

  Look for the optimal containers for your data

- Spending some time choosing your appropriate data container can be a big time saver in the long run

Summary

- Nowadays you should be aware of the memory hierarchy for getting good performance
- Leverage existing memory-efficient libraries for performing computations optimally
- Use the appropriate data containers for your different use cases

Questions?

@FrancescAlted
francesc@blosc.org

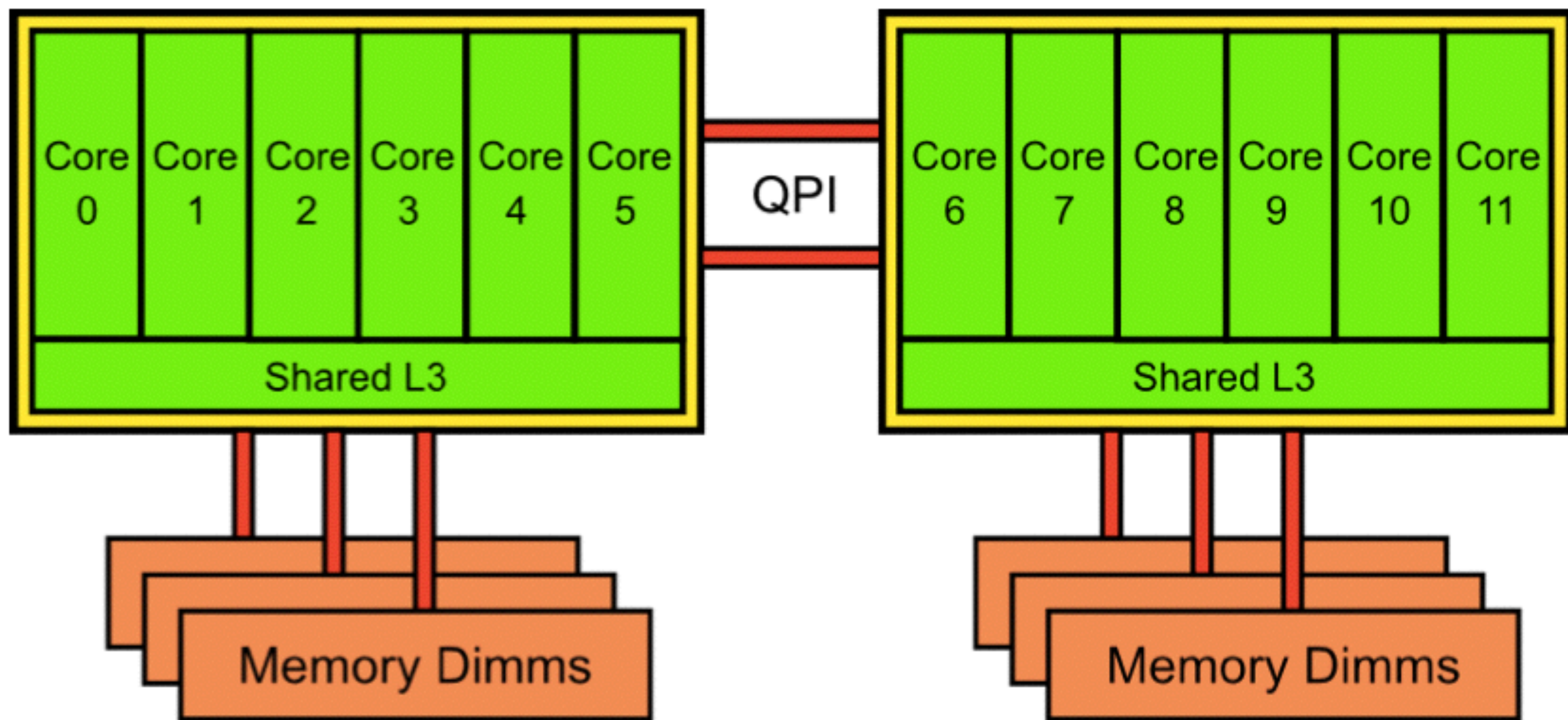
What’s Next

In the following exercises we will:

- Experiment with the numexpr library, and how it scales in a multicore machine
- Learn when your problem is CPU-bounded or memory-bounded
- Do some queries on very large datasets by using NumPy, pandas and bcolz

The ‘Big Iron’ Machine for the Exercises

Four ‘blades’ like this one:



For a grand total of 48 physical cores!