Memory Bound Computing

Francesc Alted
Freelance Consultant
http://www.blosc.org/professional-services.html

Advanced Scientific Programming in Python
Reading, UK
September, 2016

“No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be.”

— *Isaac Asimov*

“No sensible decision can be made any longer without taking into account not only the computer as it is, but the computer as it will be.”

— My own rephrasing

Overview

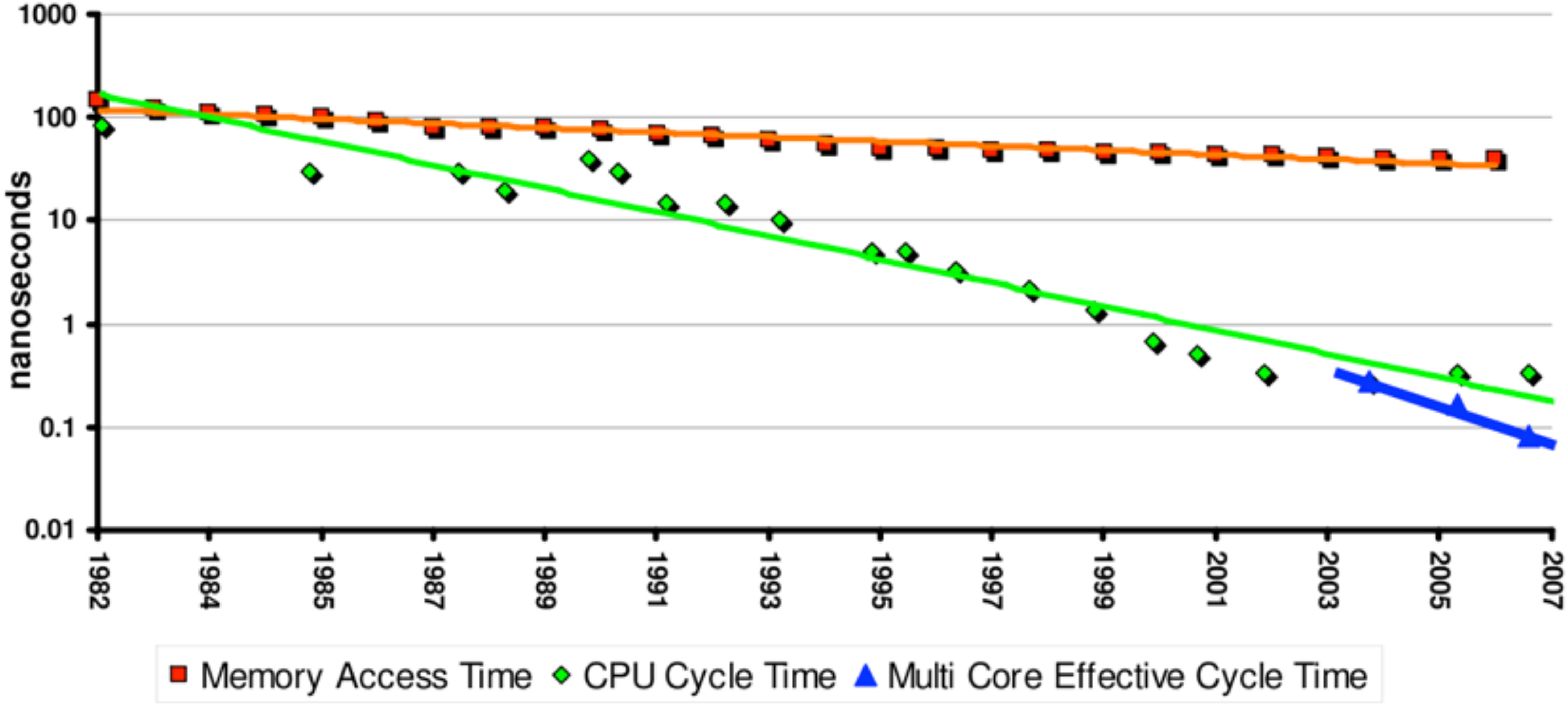
- The Starving CPU problem
- Recent trends in computer architecture
- Handling Big Data: storing and processing as much data as possible with your existing resources

“Across the industry, today’s chips are largely able to execute code faster than we can feed them with instructions and data.”

– Richard Sites, after his article
“It’s The Memory, Stupid!”,
Microprocessor Report, 10(10),1996

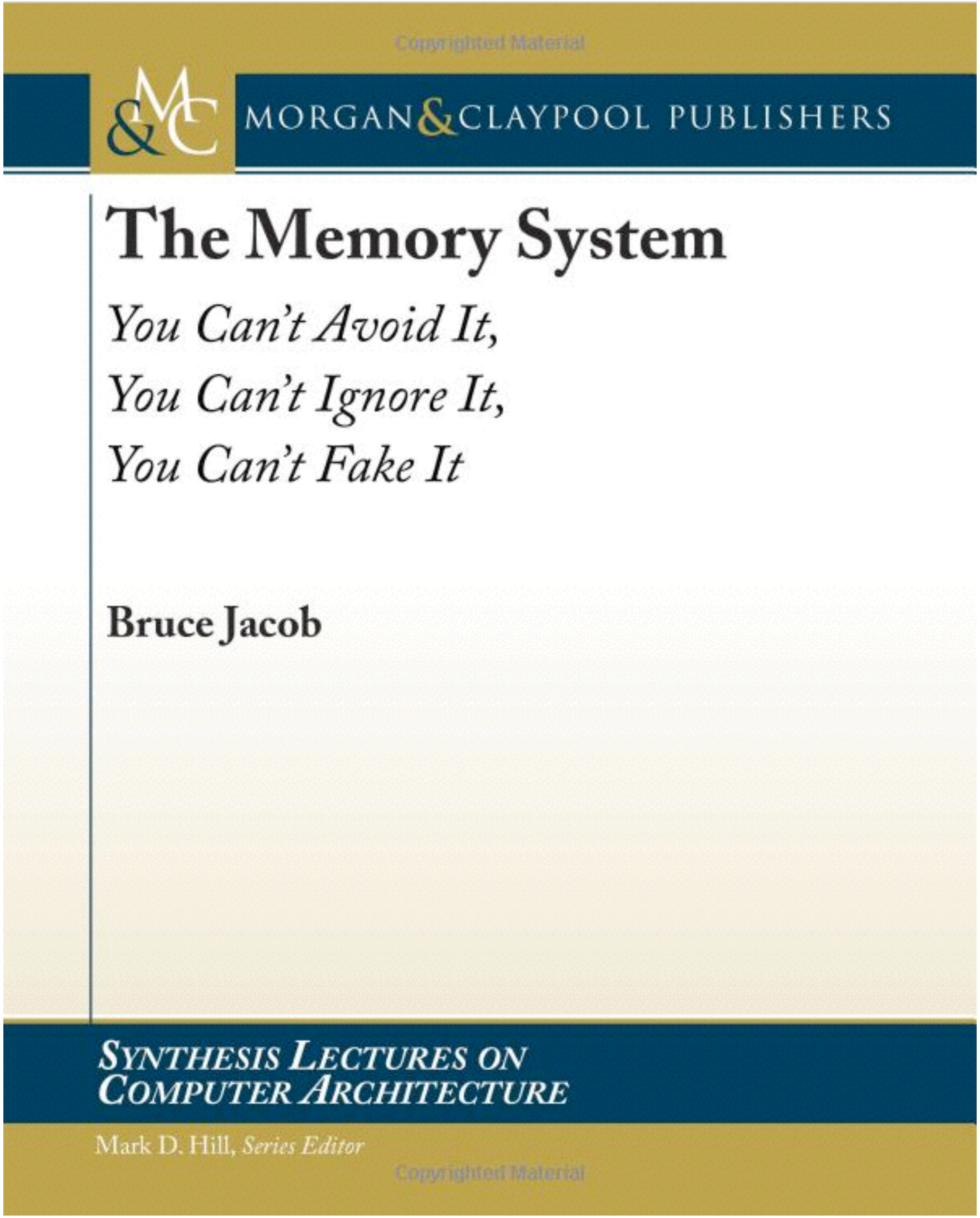
The Starving CPU Problem

Memory Access Time vs CPU Cycle Time



The gap is wide and still opening!

Book in 2009



M&C 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*

The Status of CPU Starvation in 2016

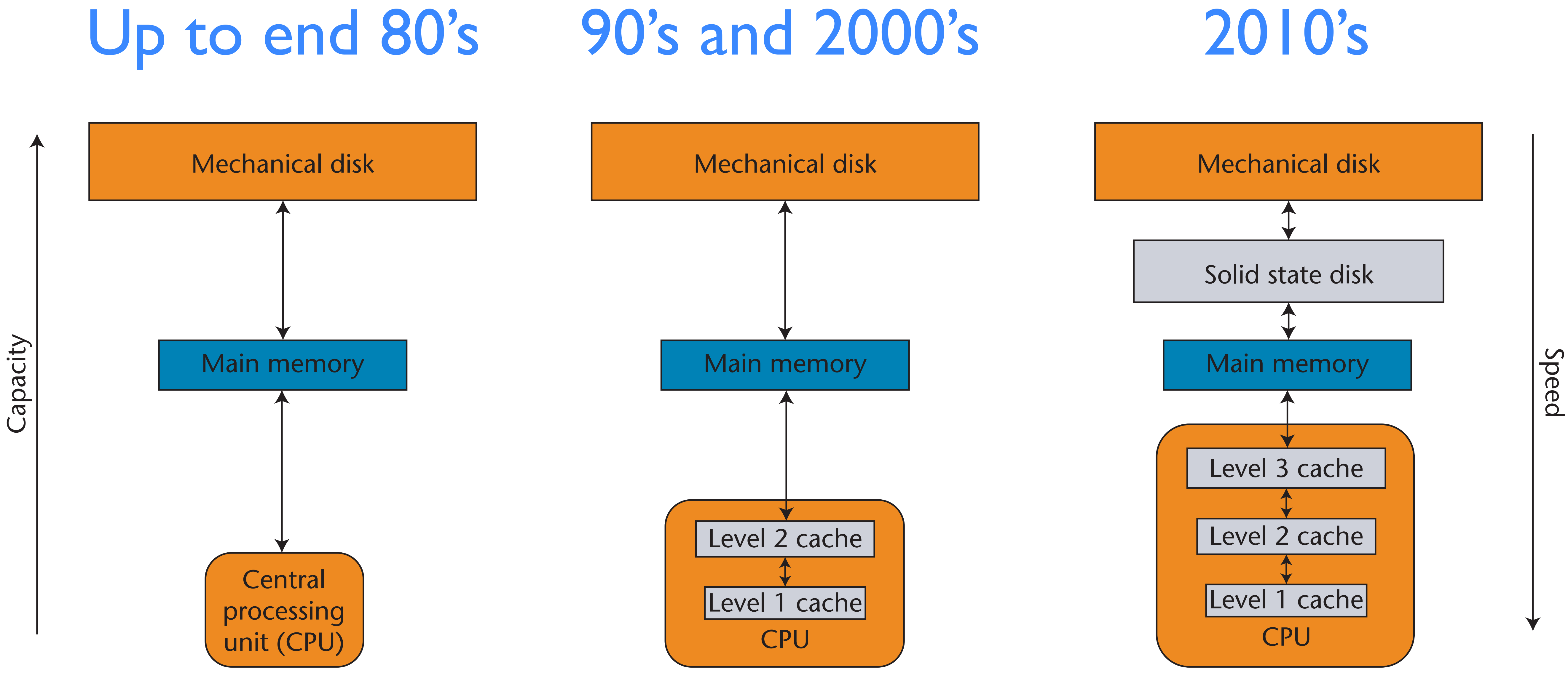
- Memory latency is much slower (between 250x and 1000x) than processors.

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

CPU Caches to the Rescue

- CPU cache latency and throughput are much better than memory
- However: the faster they run the smaller they must be (because of heat dissipation problems)

Computer Architecture Evolution



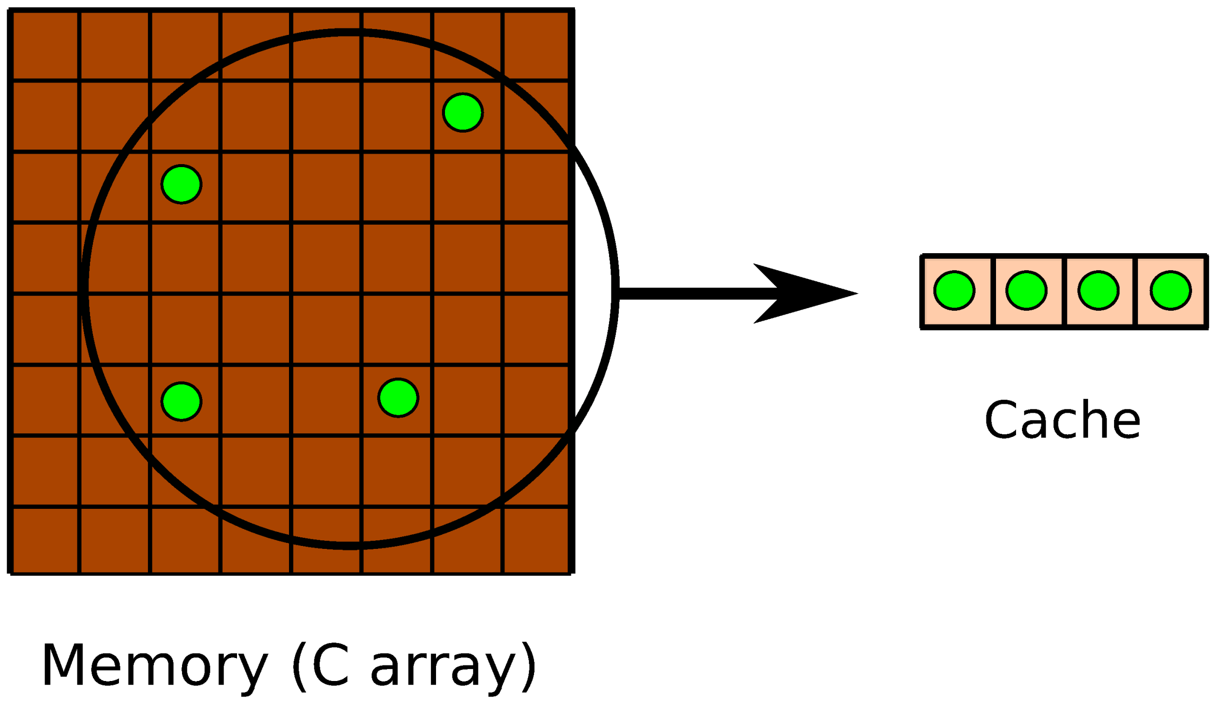
When CPU Caches Are Effective?

Mainly in a couple of scenarios:

- Time locality: when the dataset is reused
- Spatial locality: when the dataset is accessed sequentially

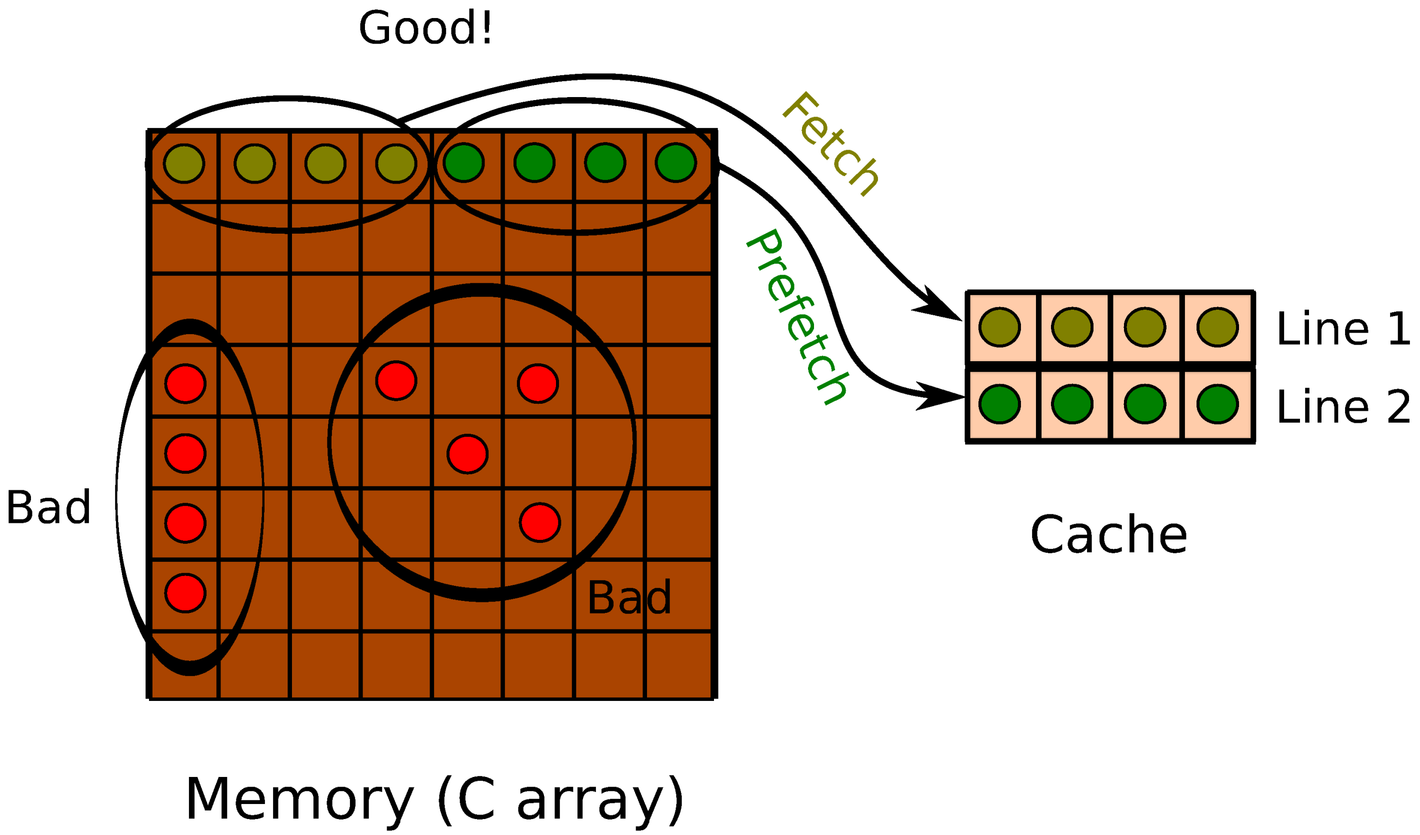
Time Locality

Parts of the dataset are reused



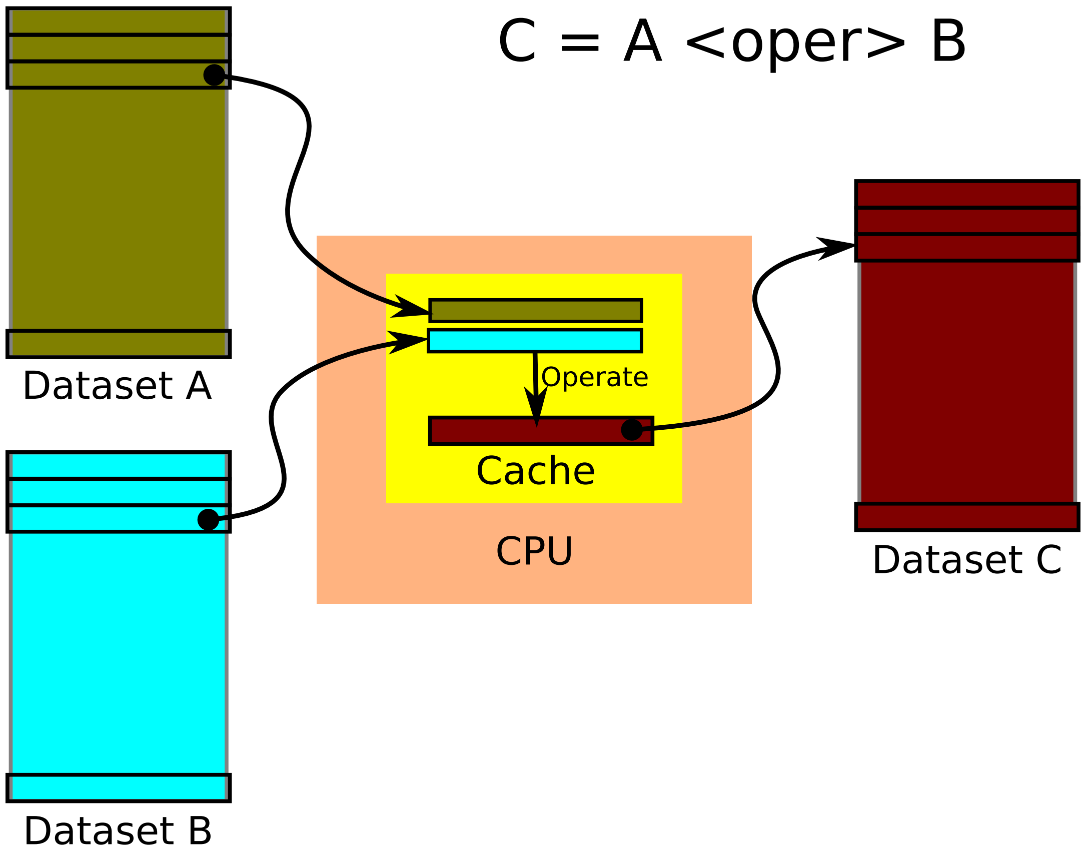
Spatial Locality

Dataset is accessed sequentially



The Blocking Technique

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

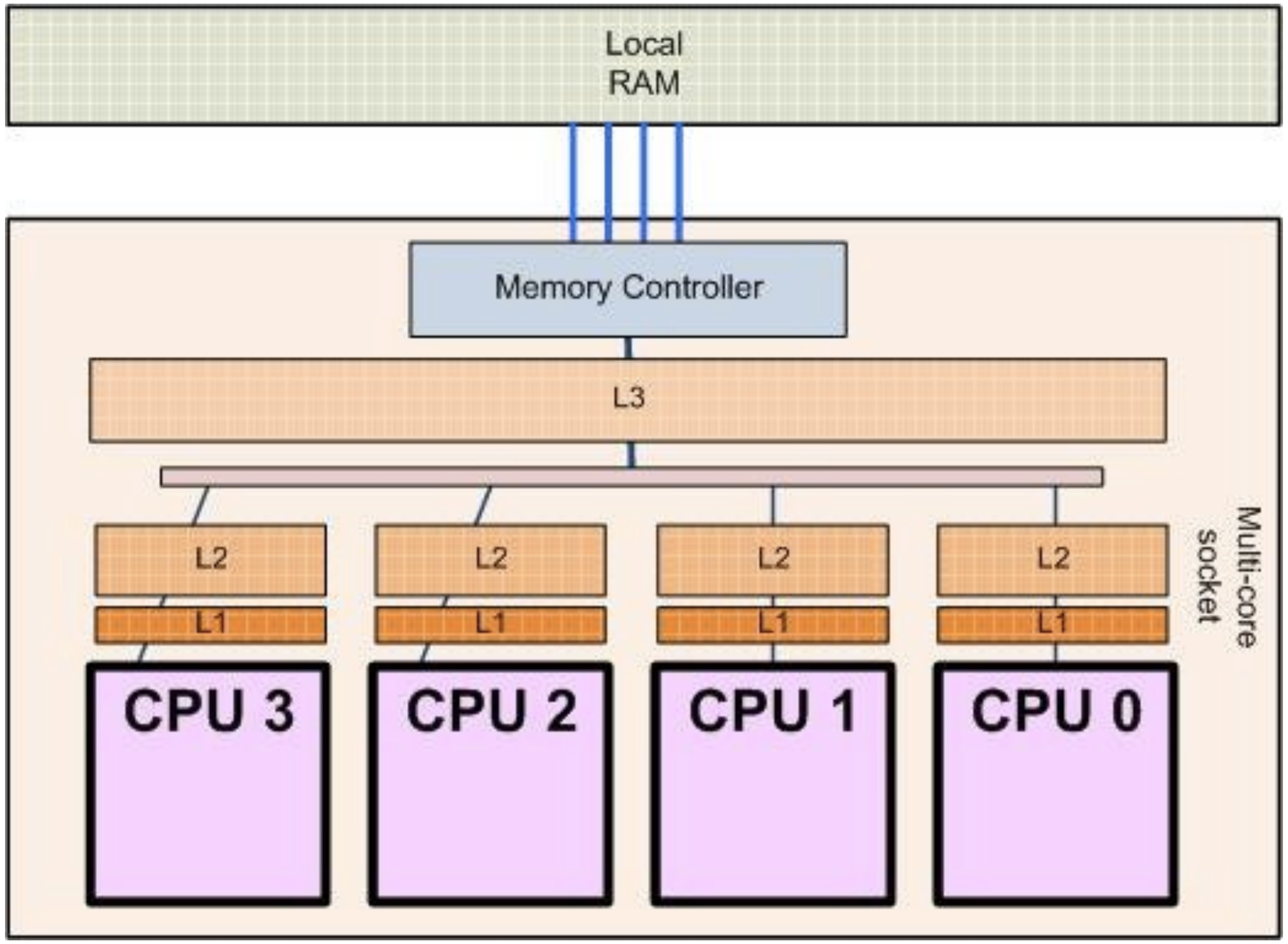
“Across the industry, today’s chips are largely able to execute code faster than we can feed them with instructions and data. There are no longer performance bottlenecks in the floating-point multiplier or in having only a single integer unit. The real design action is in memory subsystems— caches, buses,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

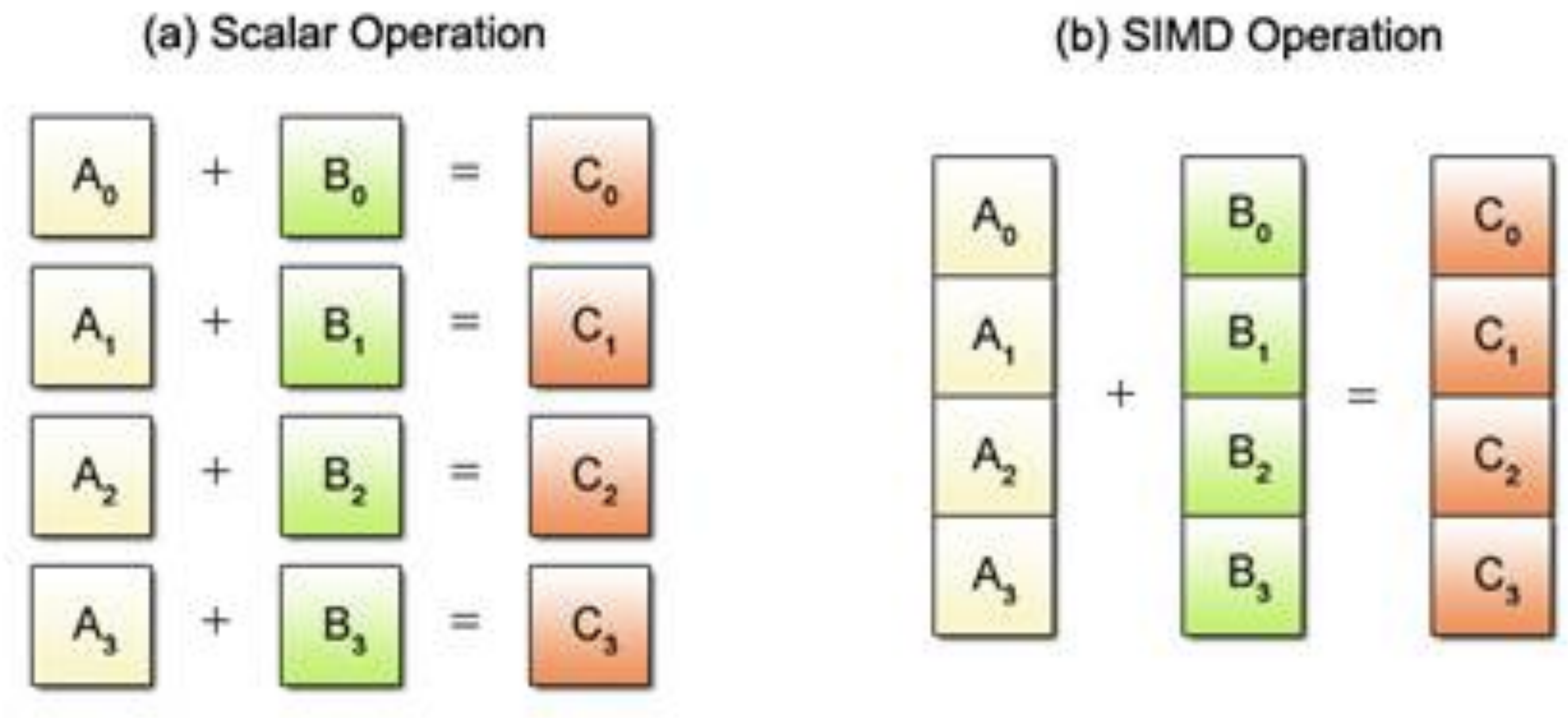
Trends in Computer Architecture

We Are In A Multicore Age



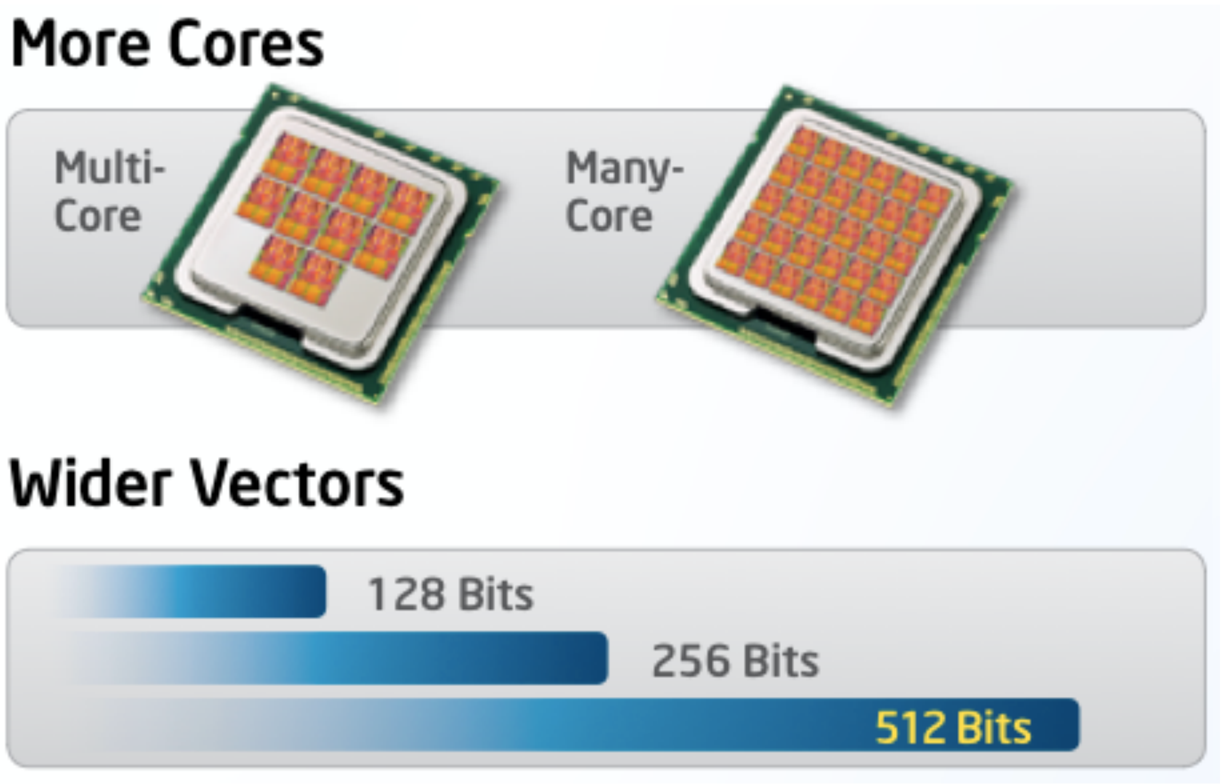
- This requires special programming measures to leverage all its potential: threads, multiprocessing

SIMD: Single Instruction, Multiple Data

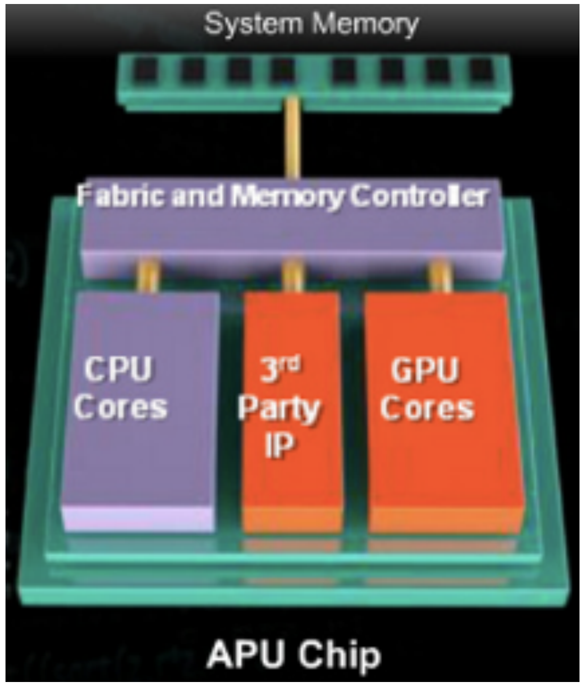


More operations in the same CPU clock

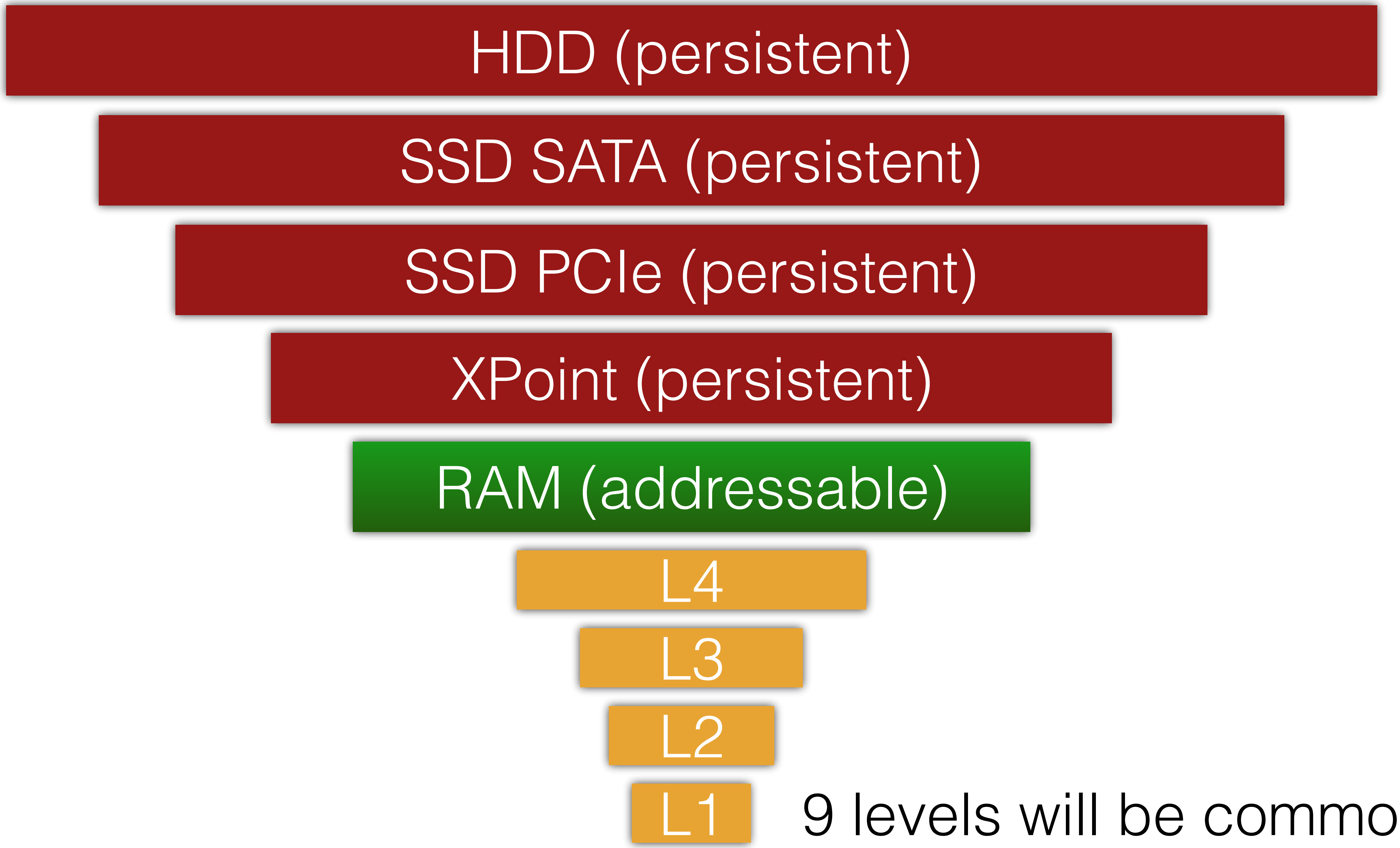
Forthcoming Trends in CPU



CPU+GPU Integration



Hierarchy of Memory
By 2018 (Educated Guess)



HDD (persistent)

SSD SATA (persistent)

SSD PCIe (persistent)

XPoint (persistent)

RAM (addressable)

L4

L3

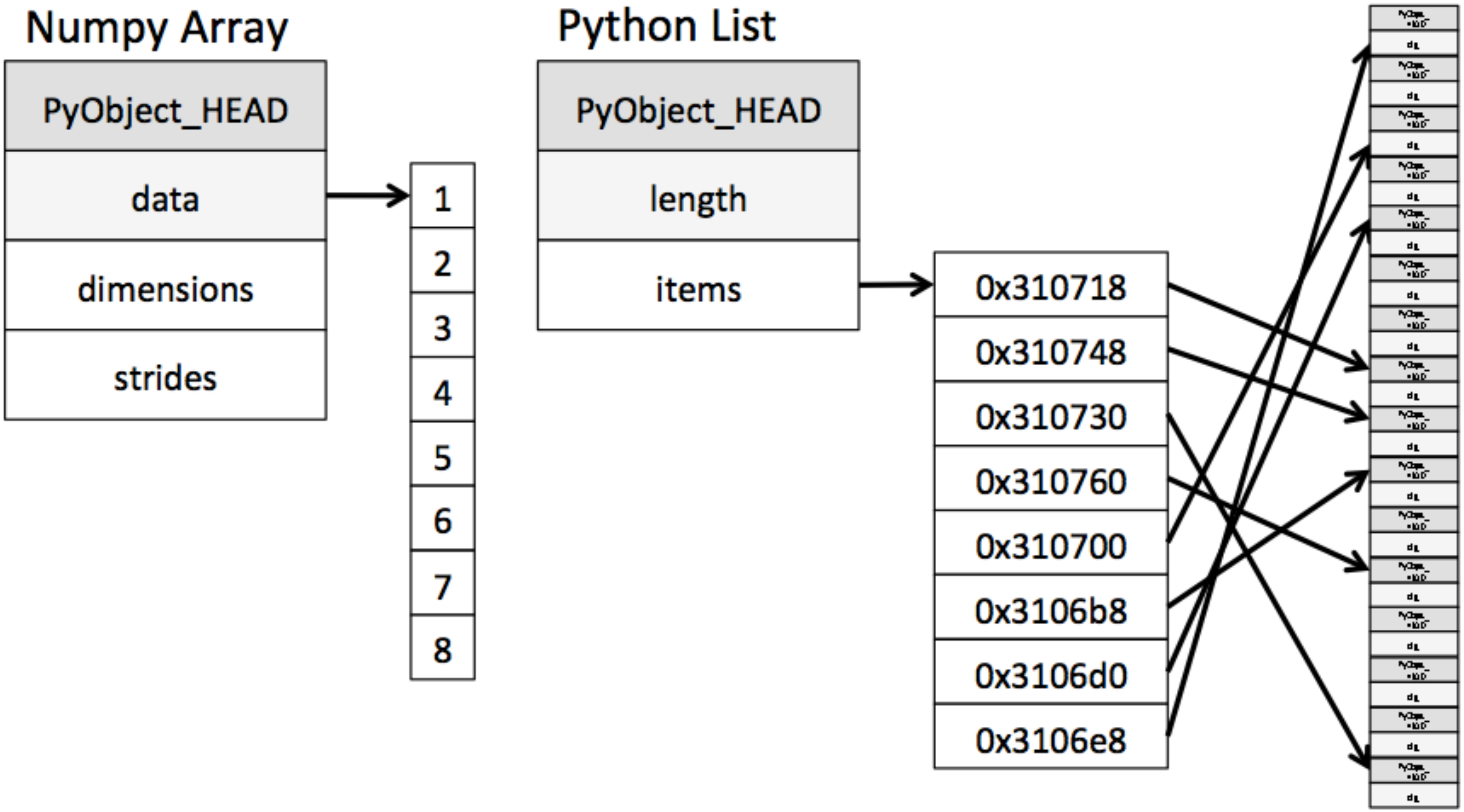
L2

L1

9 levels will be common!

Efficient Data Containers for In-Memory Storage

Which Structure Is More Efficient for Computing?



Numpy Array

PyObject_HEAD

data

dimensions

strides

1

2

3

4

5

6

7

8

Python List

PyObject_HEAD

length

items

0x310718

0x310748

0x310730

0x310760

0x310700

0x3106b8

0x3106d0

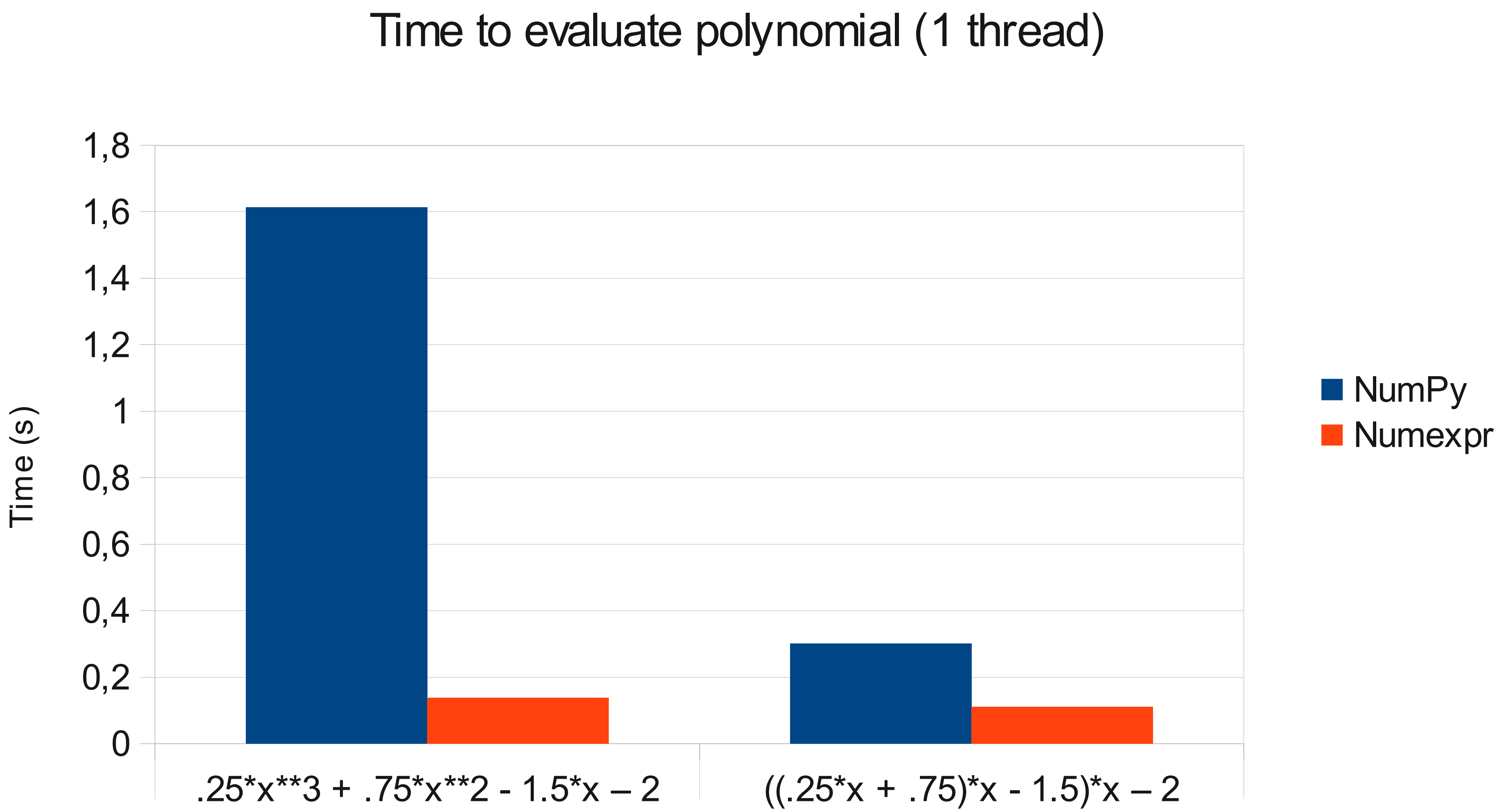
0x3106e8

Attr: Jake Vanderplas

numexpr

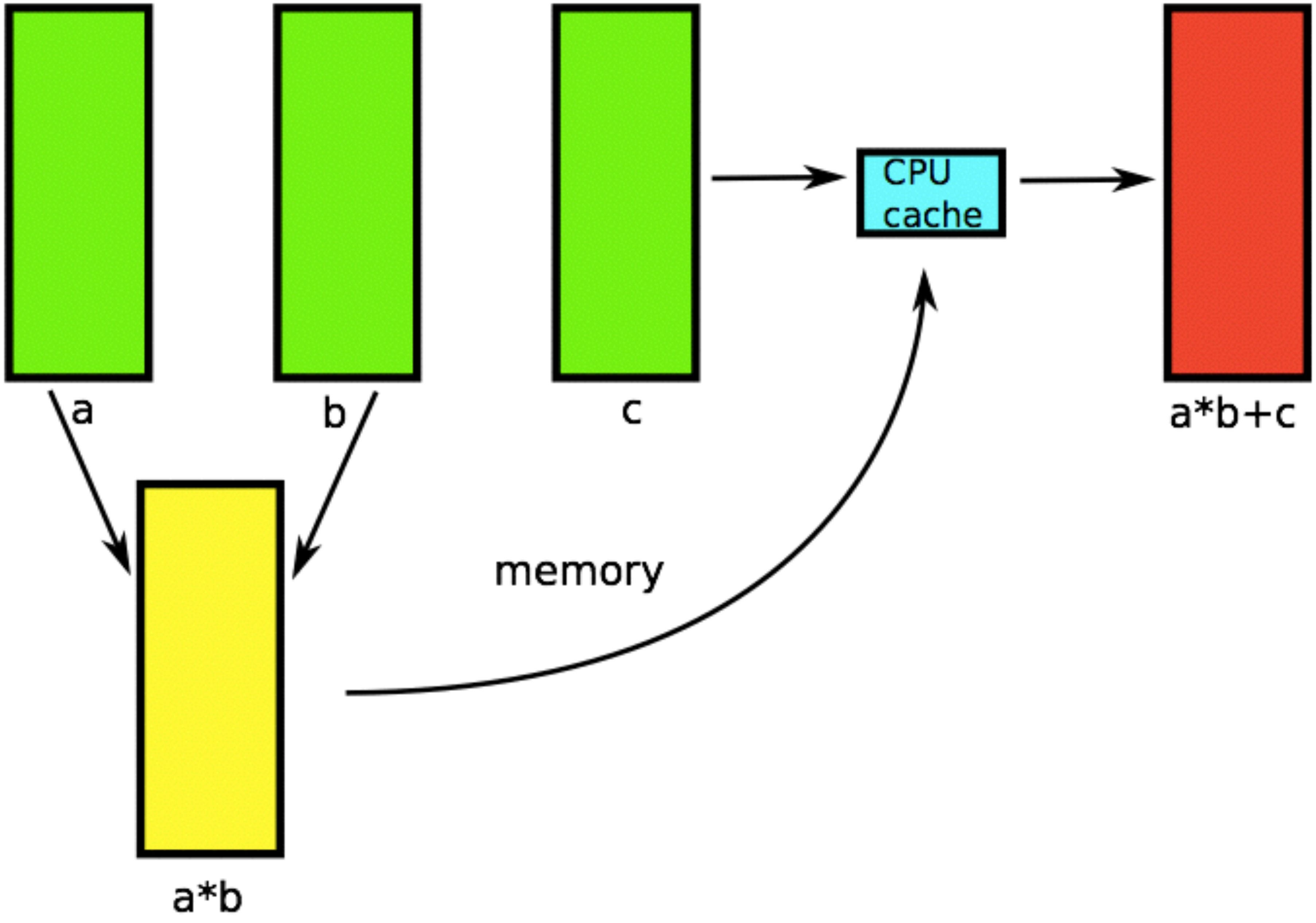
- numexpr is a computational engine for NumPy that makes a sensible use of the memory hierarchy for better performance
- Other libraries normally use numexpr when it is time to accelerate large computations
- PyTables, pandas and bcolz (among others) can all leverage numexpr

Time To Answer Pending Questions



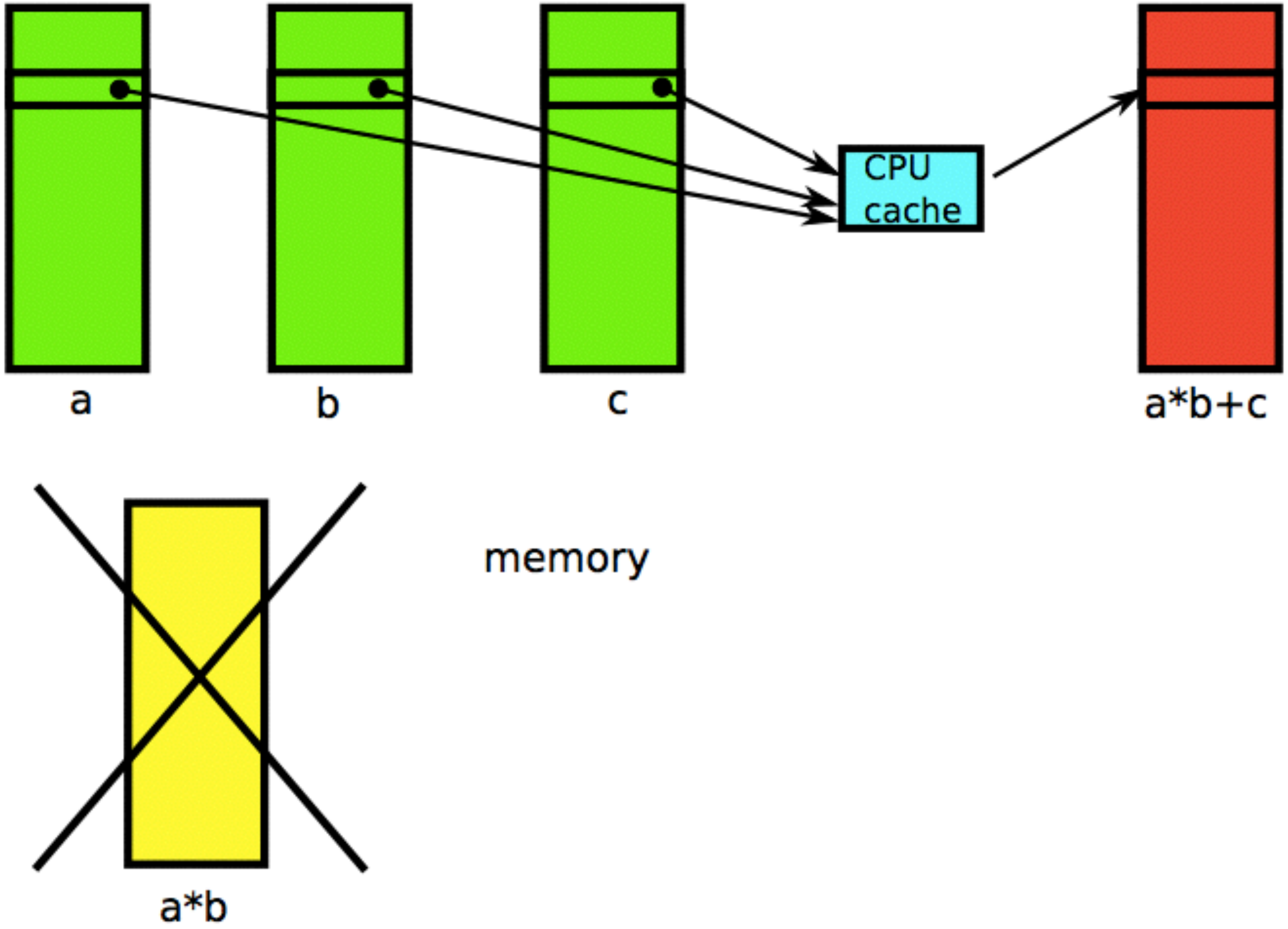
Computing with NumPy

Temporaries go to memory



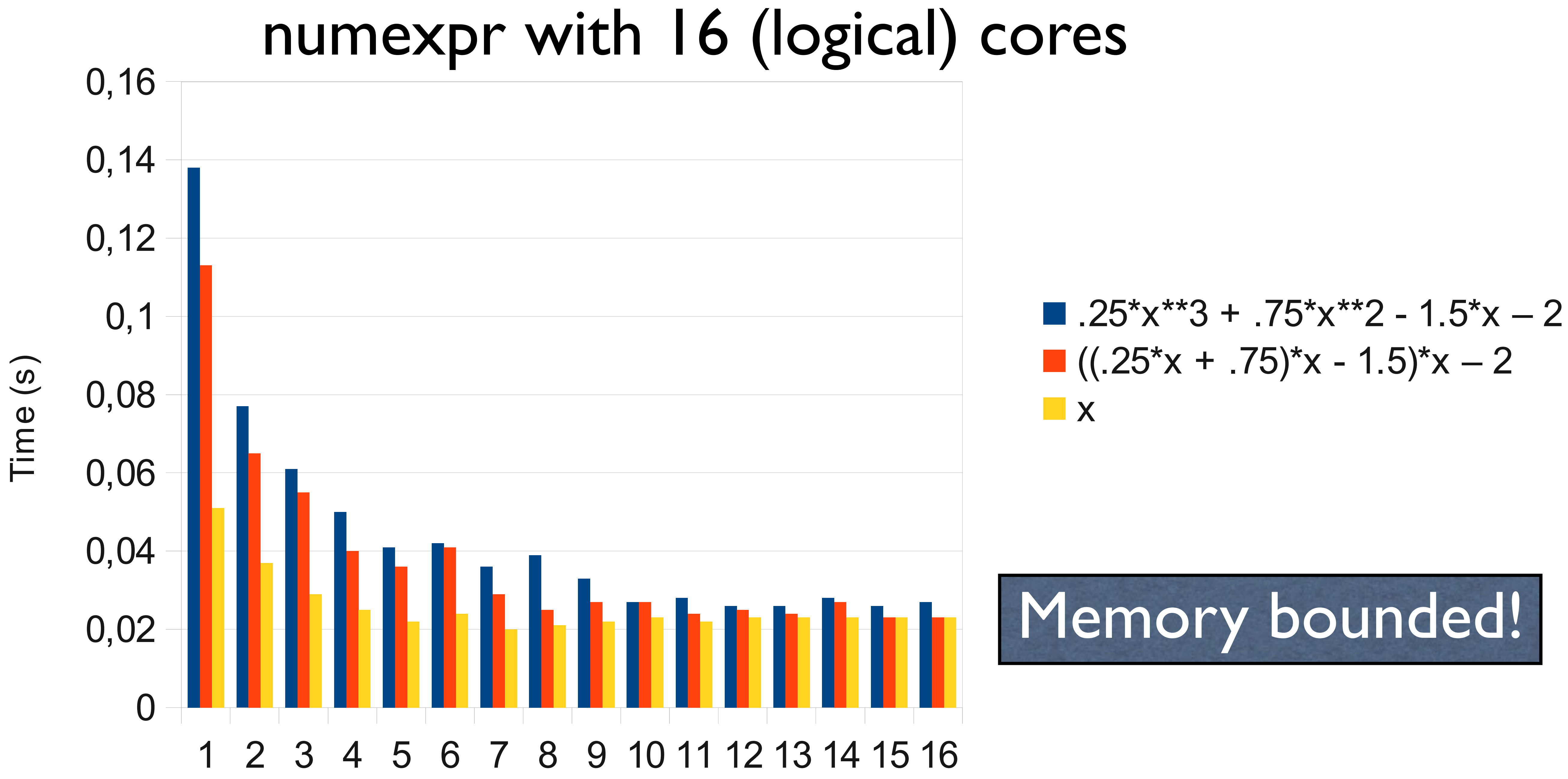
Computing with numexpr

Temporaries stay in cache



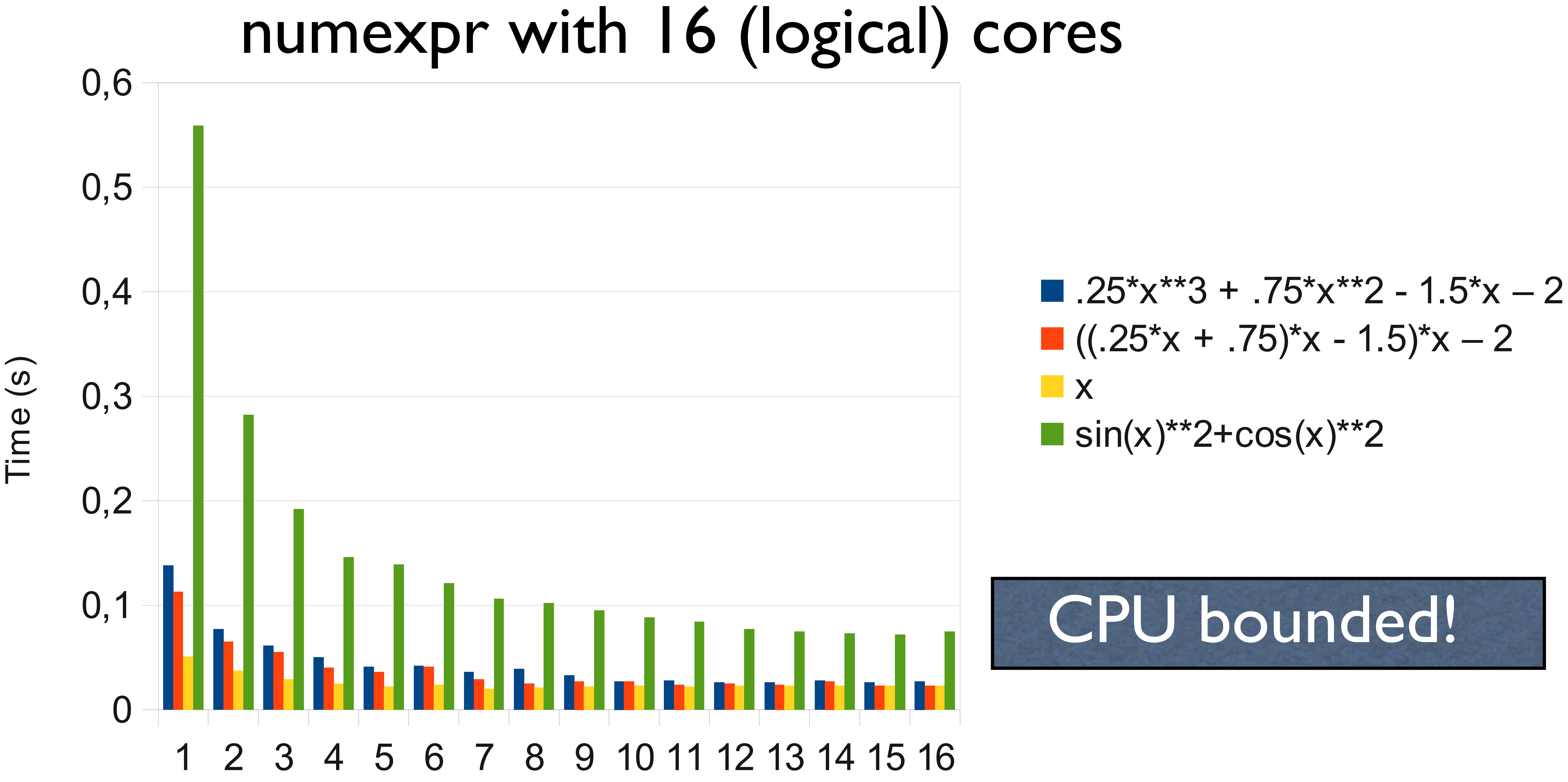
Multithreaded numexpr and Beyond: Numba

numexpr Allows Multithreading for Free



Memory bounded!

Transcendental Functions



CPU bounded!

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]
```

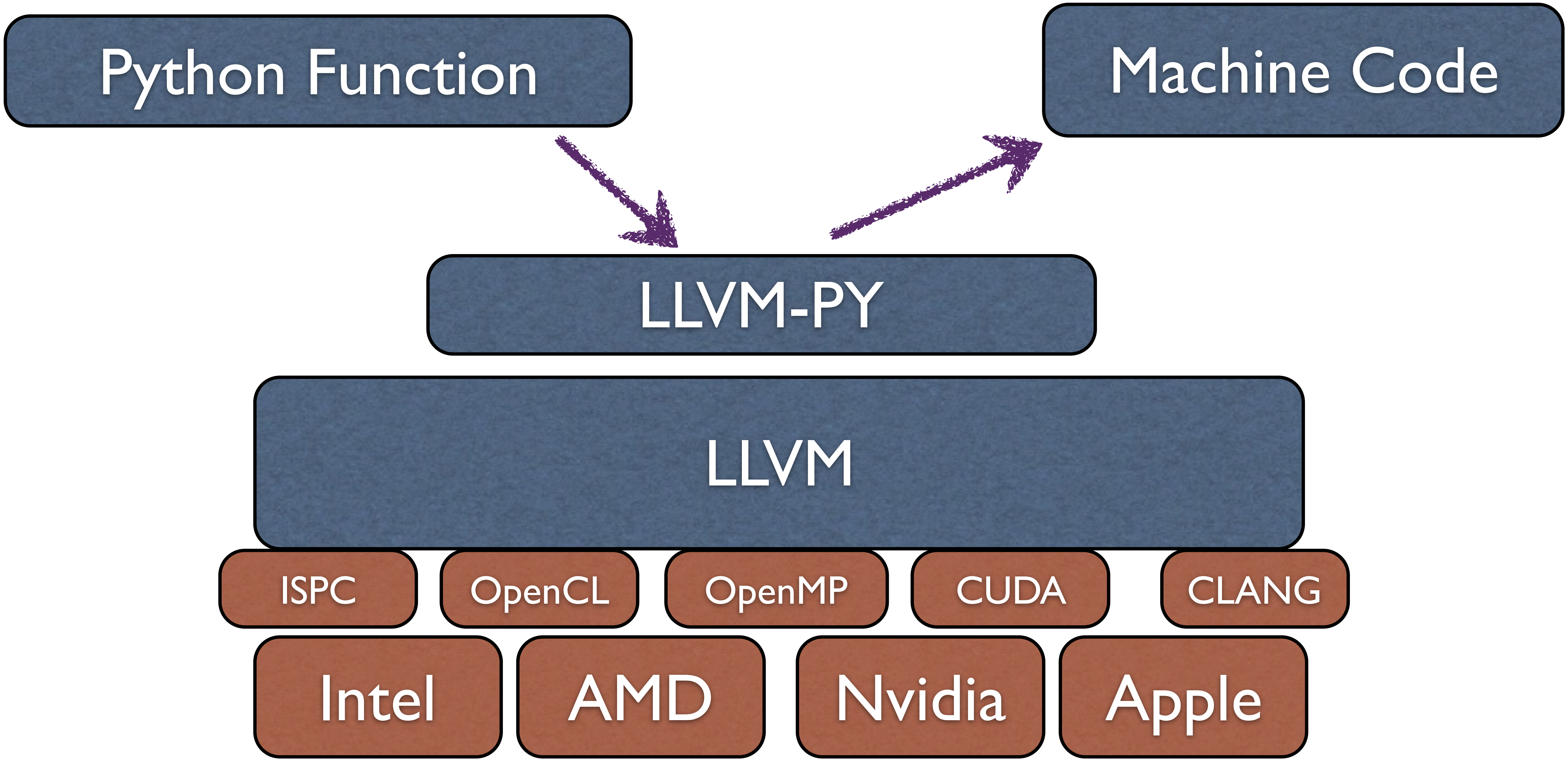
- In particular, it cannot deal with things like:

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

Numba: Overcoming numexpr Limitations

- Numba is a JIT that can translate a subset of the Python language into machine code
- It uses LLVM infrastructure behind the scenes
- Can achieve similar or better performance than numexpr, but with more flexibility

How Numba works


Python Function

Machine Code

LLVM-PY

LLVM

ISPC

OpenCL

OpenMP

CUDA

CLANG

Intel

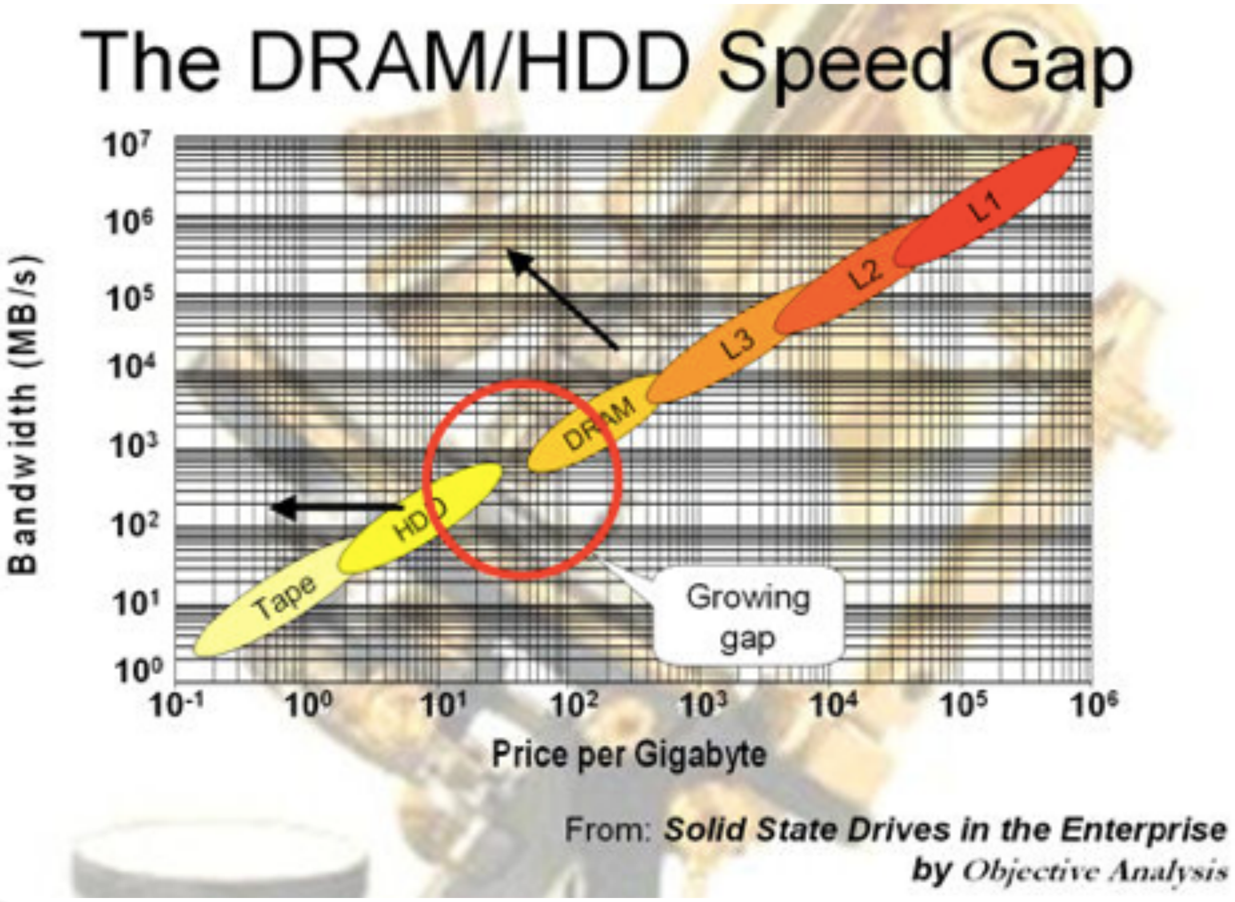
AMD

Nvidia

Apple

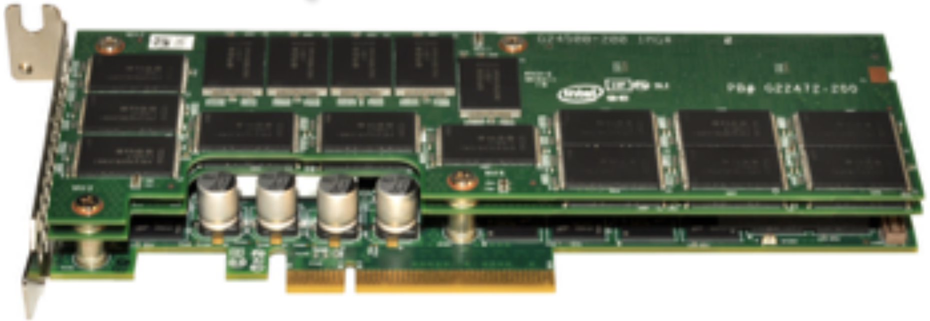
Trends in Computer Storage

The growing gap between DRAM and HDD is facilitating the introduction of new SDD devices



The DRAM/HDD Speed Gap

From: ***Solid State Drives in the Enterprise***
by *Objective Analysis*



PCIe SSD

M.2 SSD

BGA SSD

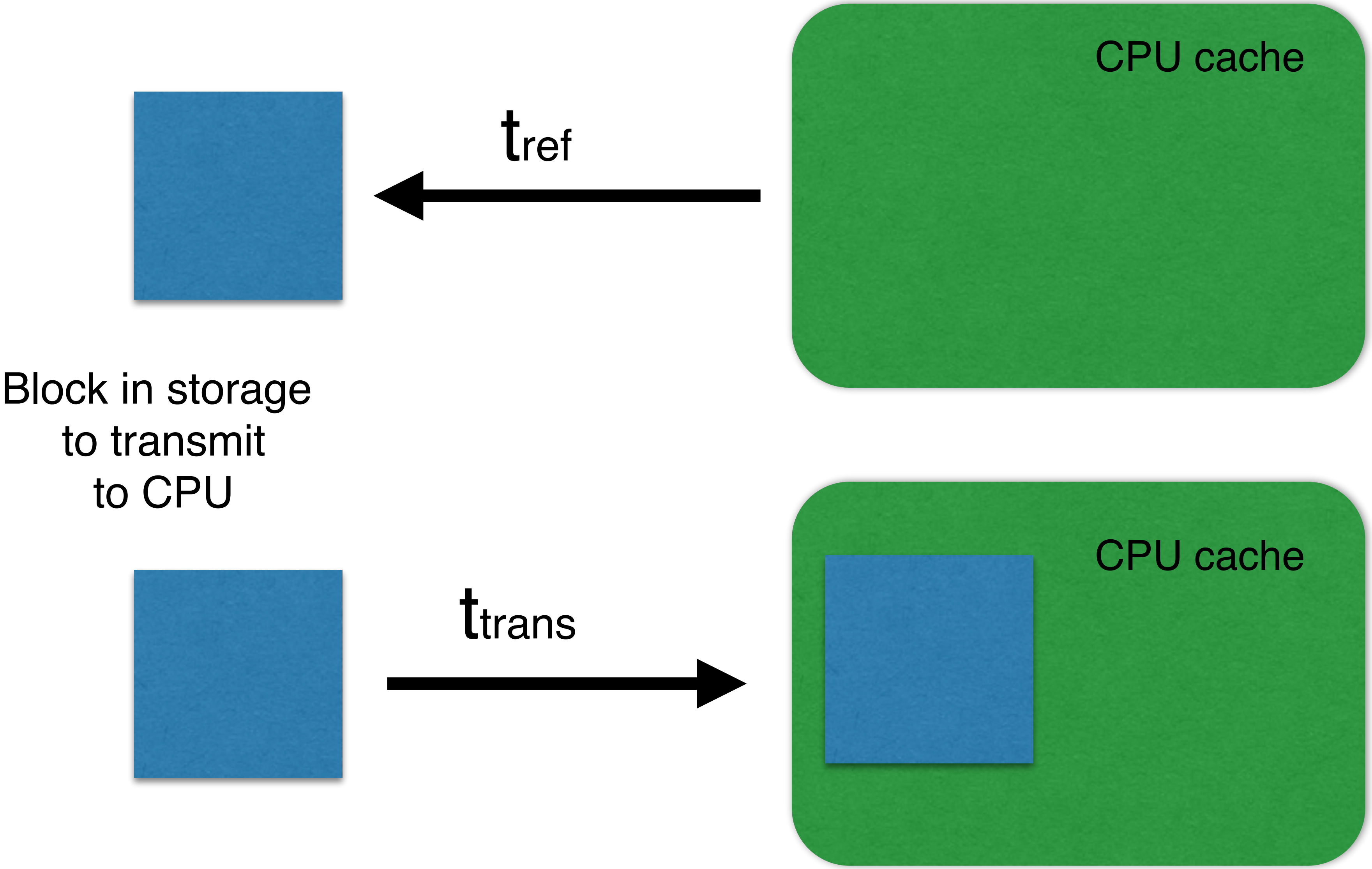
Latency Numbers Every Programmer Should Know

```
Latency Comparison Numbers
--------------------------
L1 cache reference                            0.5 ns
Branch mispredict                             5   ns
L2 cache reference                            7   ns             14x L1 cache
Mutex lock/unlock                            25   ns
Main memory reference                       100   ns             20x L2 cache, 200x L1 cache
Read 4K randomly from memory              1,000   ns    0.001 ms
Compress 1K bytes with Zippy              3,000   ns
Send 1K bytes over 1 Gbps network        10,000   ns    0.01 ms
Read 4K randomly from SSD*              150,000   ns    0.15 ms
Read 1 MB sequentially from memory      250,000   ns    0.25 ms
Round trip within same datacenter       500,000   ns    0.5  ms
Read 1 MB sequentially from SSD*      1,000,000   ns    1    ms  4X memory
Disk seek                            10,000,000   ns   10    ms  20x datacenter roundtrip
Read 1 MB sequentially from disk     20,000,000   ns   20    ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA     150,000,000   ns  150    ms
```

Source: Jeff Dean and Peter Norvig (Google), with some additions

http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Reference Time vs Transmission Time



$t_{ref} \sim= t_{trans}$ => optimizes memory access

Not All Storage Layers Are Created Equal

Memory: t_{ref}: 100 ns / t_{trans} (**1 KB**): ~100 ns

Solid State Disk: t_{ref}: 10 us / t_{trans} (**4 KB**): ~10 us

Mechanical Disk: t_{ref}: 10 ms / t_{trans} (**1 MB**): ~10 ms

The slower the media, the larger the block that is worth to transmit

But essentially, a blocked data access is mandatory for speed!

We Need More Data Blocking In Our Infrastructure!

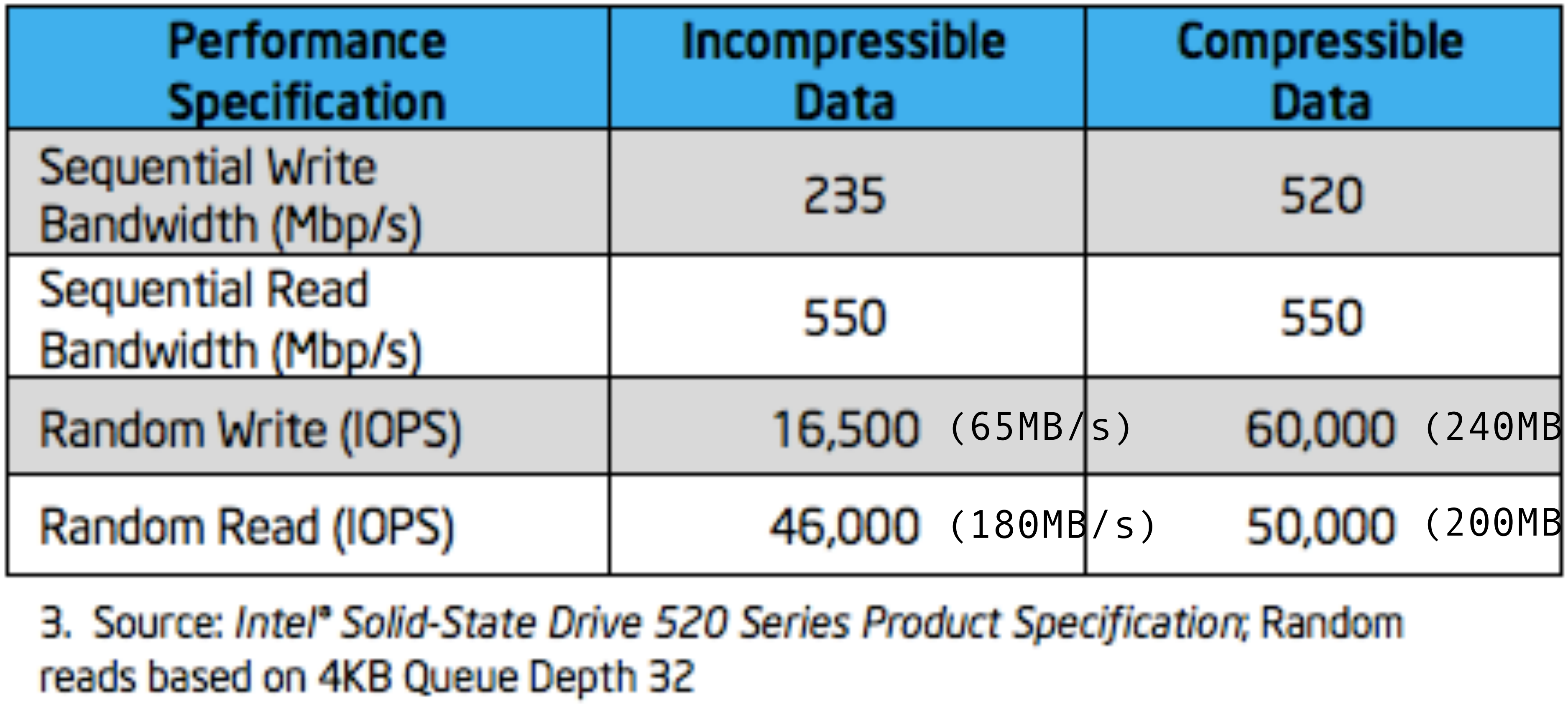
- Not many data containers leveraging the blocking technique exist yet, but a handful (e.g. HDF5, bcolz or zarr) do

- With blocked access we can use persistent media (disk) as it is ephemeral (memory) and the other way around -> independency of media!

- **No silver bullet**: we won't be able to find a single container that makes everybody happy; it's all about tradeoffs

Can We Get Better Bandwidth Than Hardware Allows?

Compression for Random & Sequential Access in SSDs

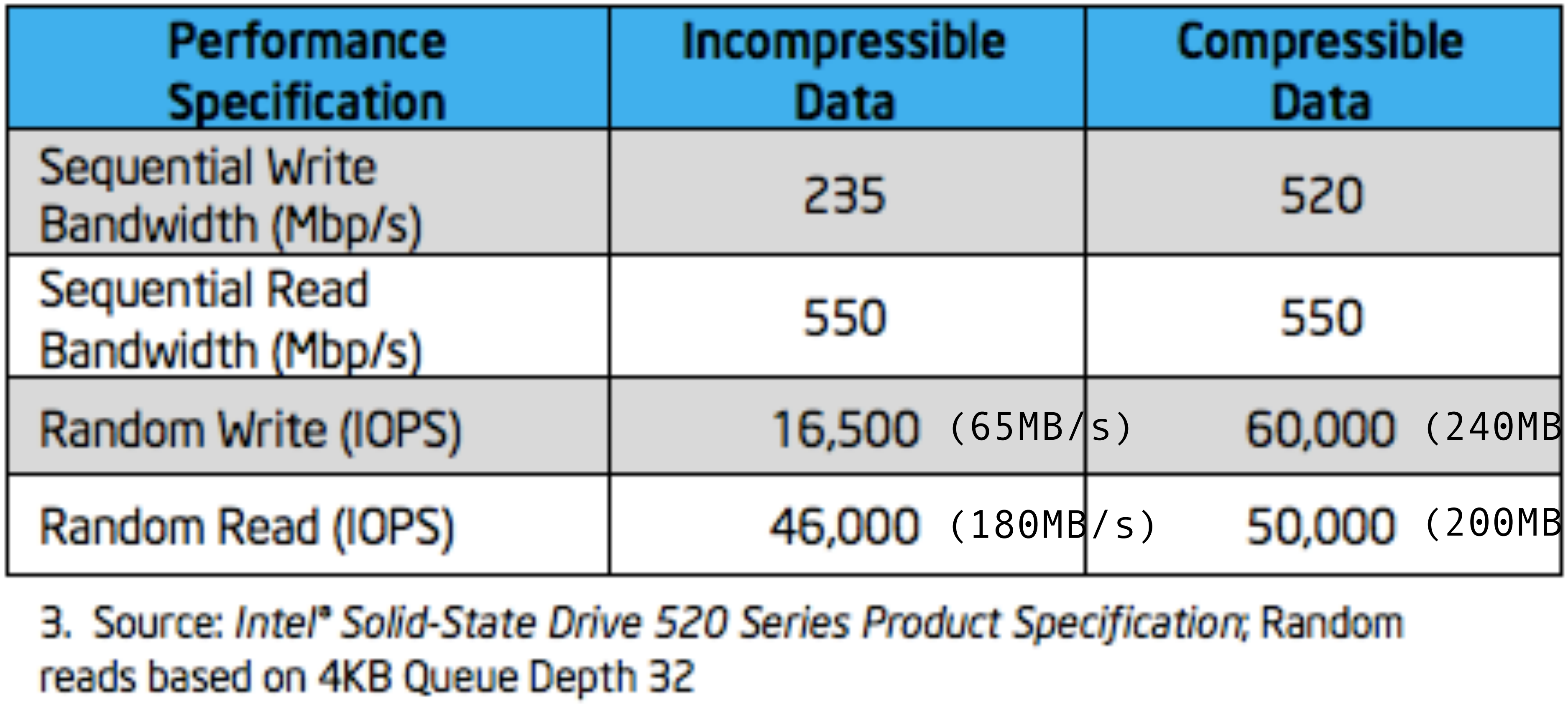


Performance Specification	Incompressible Data	Compressible Data
Sequential Write Bandwidth (Mbp/s)	235	520
Sequential Read Bandwidth (Mbp/s)	550	550
Random Write (IOPS)	16,500 (65MB/s)	60,000 (240MB/s)
Random Read (IOPS)	46,000 (180MB/s)	50,000 (200MB/s)

3. Source: *Intel® Solid-State Drive 520 Series Product Specification*; Random reads based on 4KB Queue Depth 32

- Compression does help performance!

Compression for Random & Sequential Access in SSDs



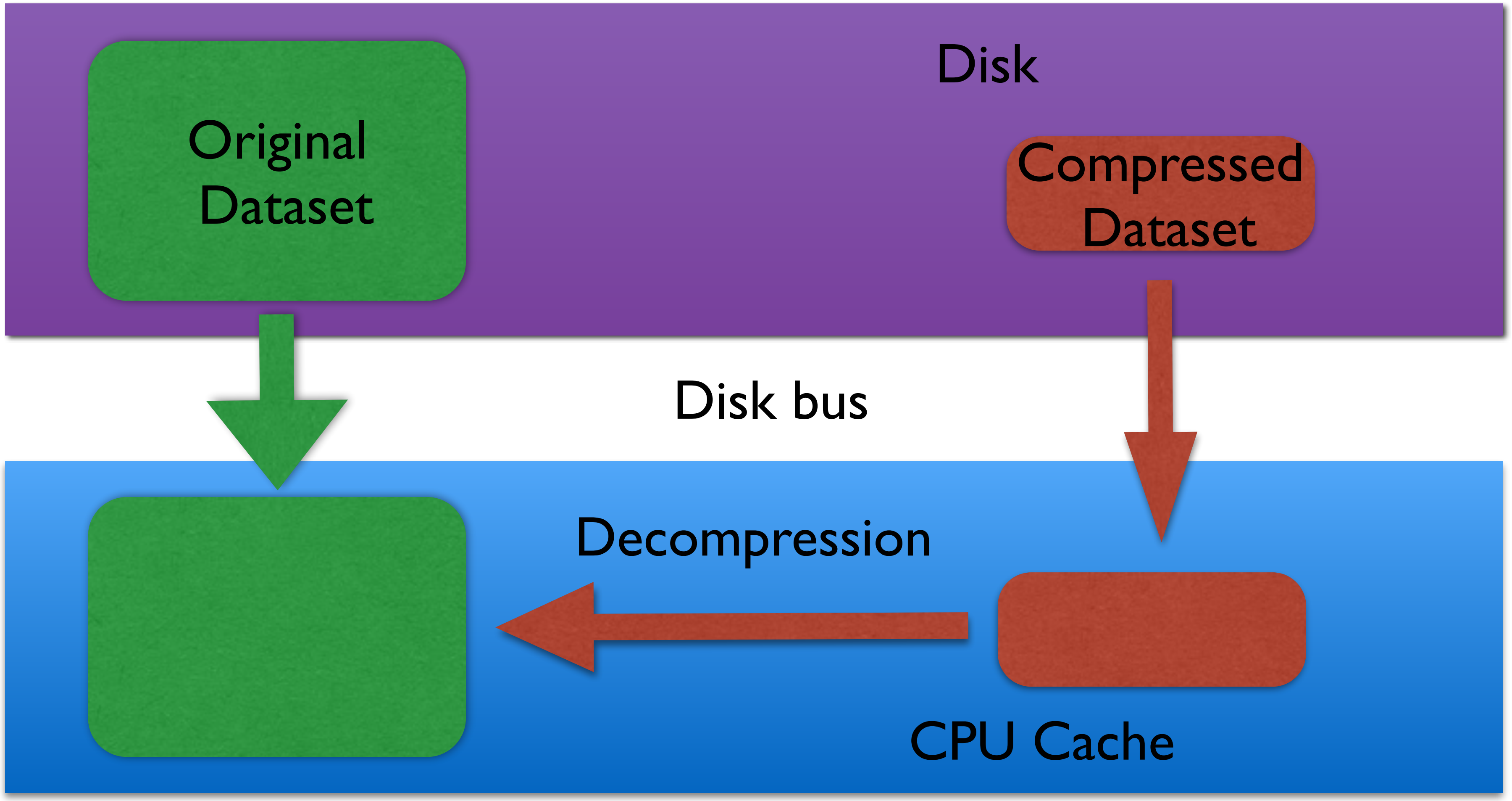
Performance Specification	Incompressible Data	Compressible Data
Sequential Write Bandwidth (Mbp/s)	235	520
Sequential Read Bandwidth (Mbp/s)	550	550
Random Write (IOPS)	16,500 (65MB/s)	60,000 (240MB/s)
Random Read (IOPS)	46,000 (180MB/s)	50,000 (200MB/s)

3. Source: *Intel® Solid-State Drive 520 Series Product Specification*; Random reads based on 4KB Queue Depth 32

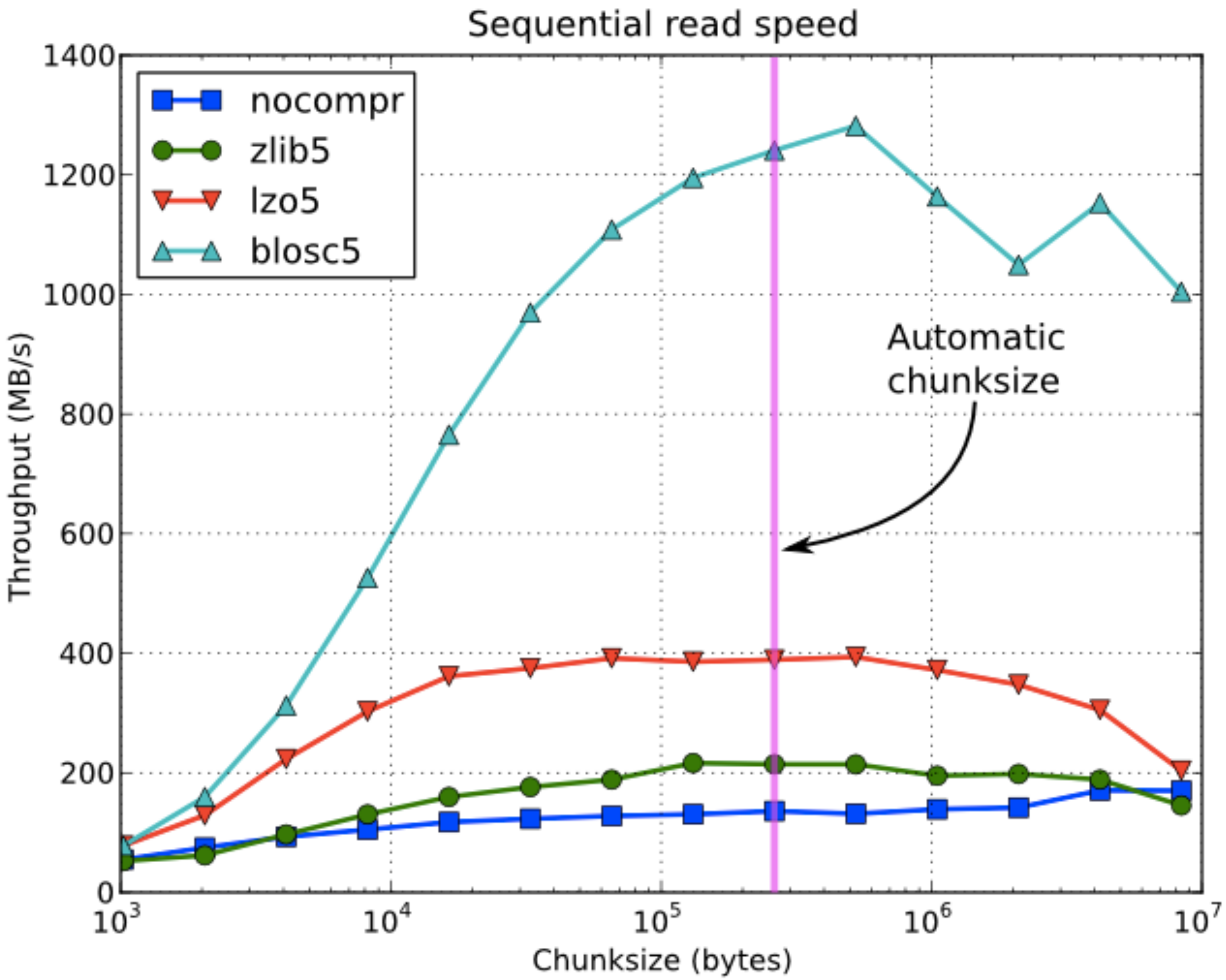
- Compression does help performance!
- However, limited by SATA bandwidth

Leveraging Compression Straight To CPU

Less data needs to be transmitted to the CPU

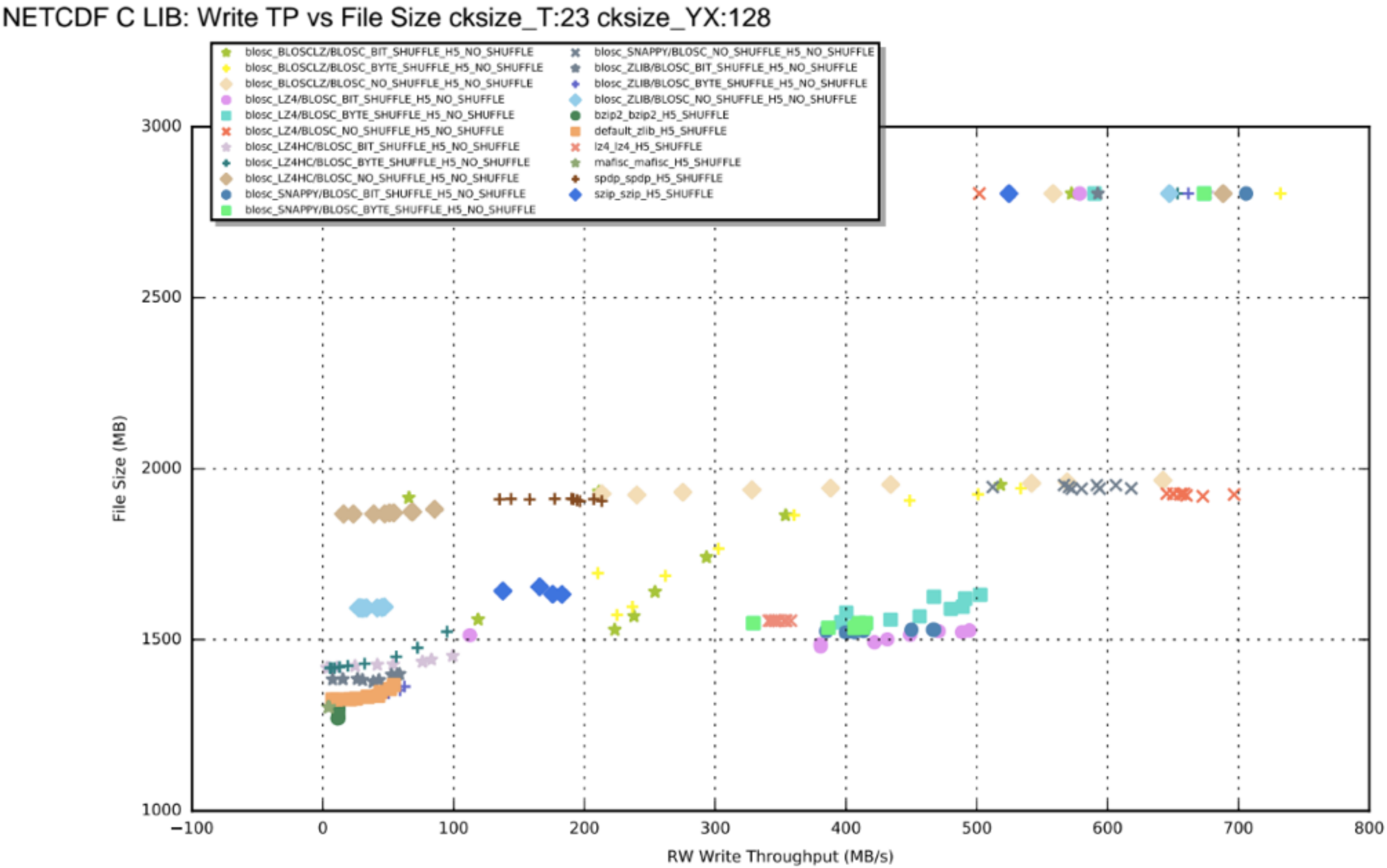


Transmission + decompression faster than direct transfer?



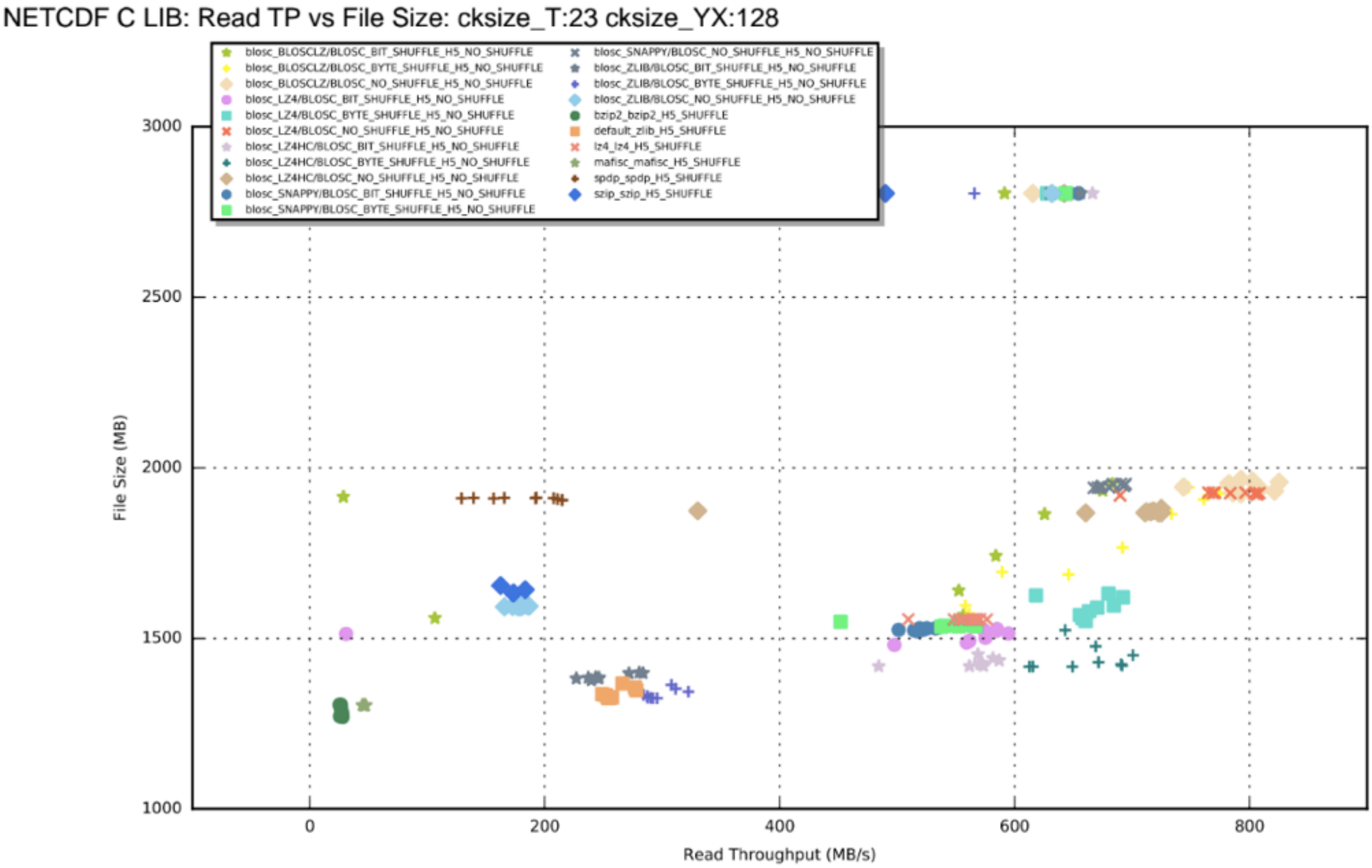
When we have a fast enough compressor
we can get rid of the limitations of the bus bandwidth.

How to get maximum compression performance?



Thanks to: Rui Yang, Pablo Larraondo (NCI Australia)

Example with actual data (satellite images):
Blosc compression does not degrade I/O performance



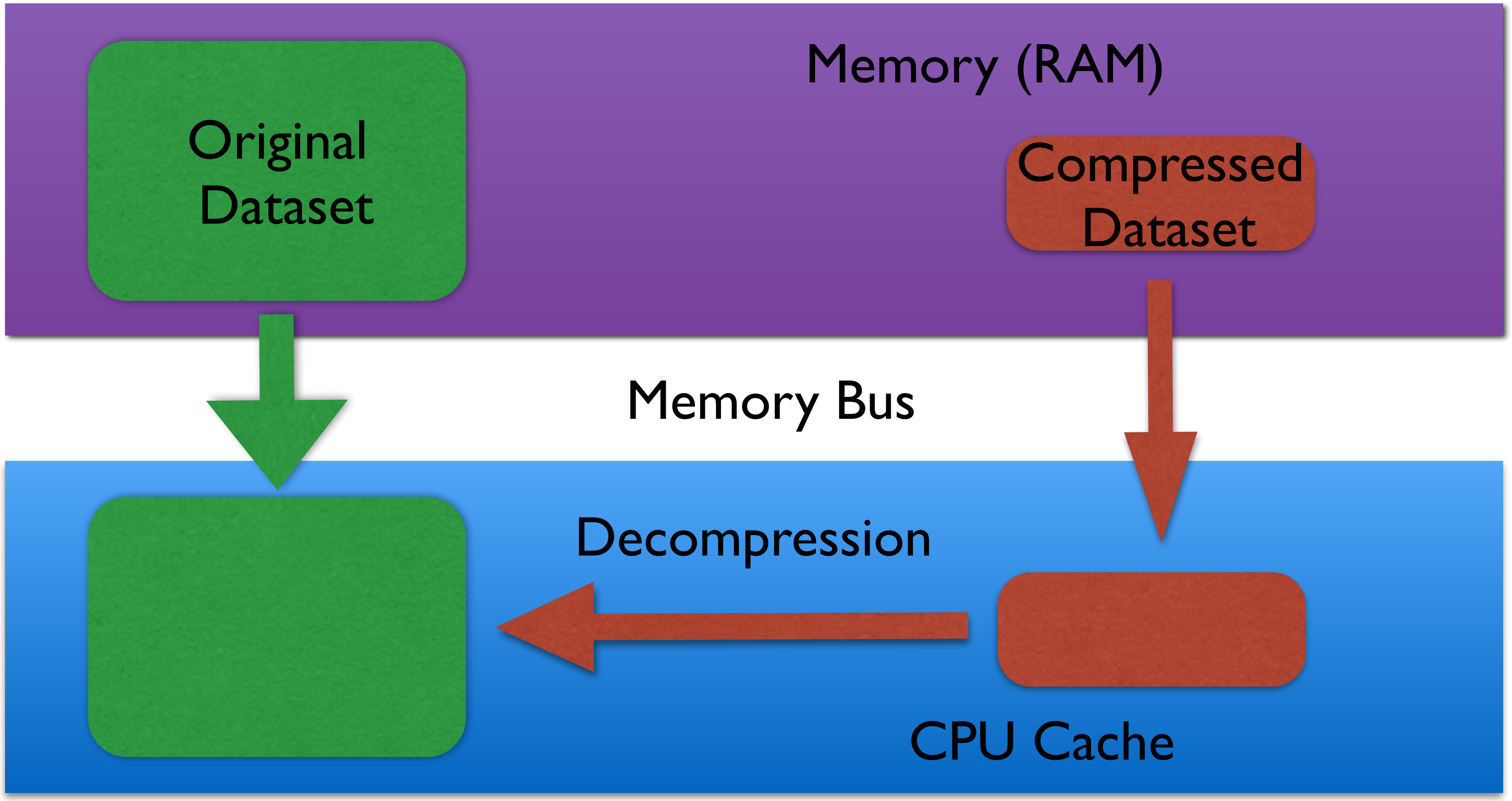
Thanks to: Rui Yang, Pablo Larraondo (NCI Australia)

Reading satellite images:
Blosc decompression accelerates I/O

Can CPU-based Compression Alleviate The Memory Bottleneck?

Improving RAM Speed?

Less data needs to be transmitted to the CPU



Transmission + decompression faster than direct transfer?

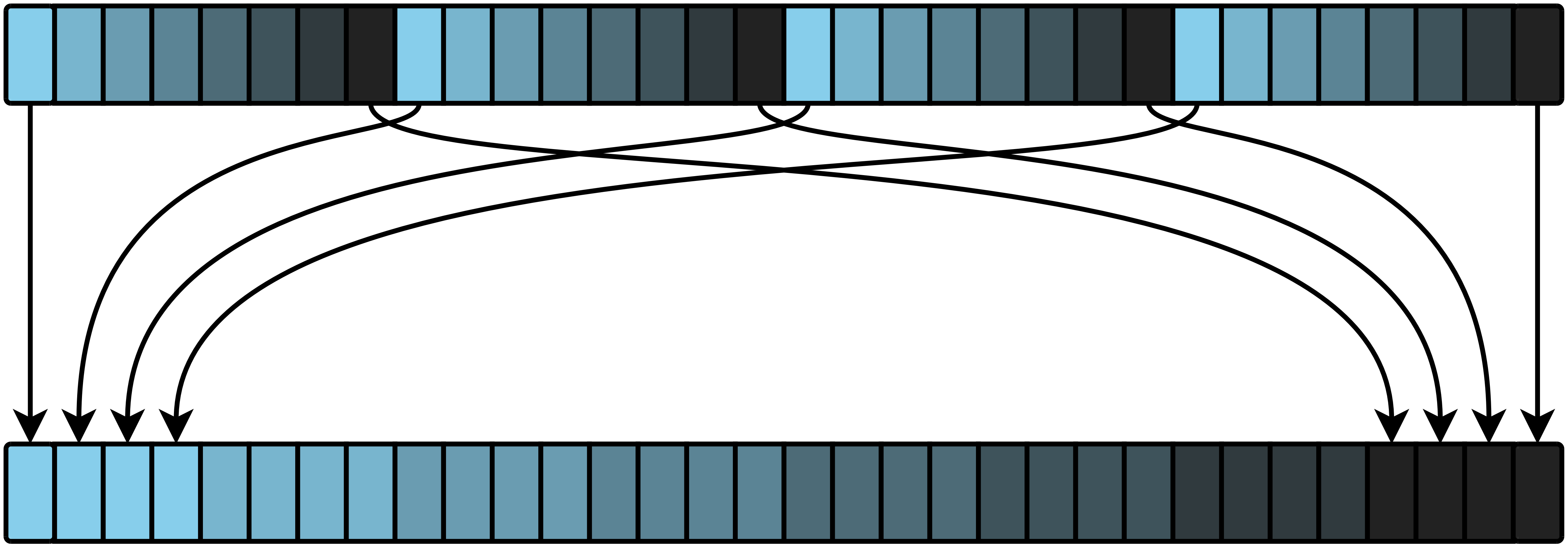
We can try, but certain conditions must be met

- We must follow the principles we have seen:
  - Data Containers leveraging the blocking technique (better cache usage)
  - Using (extremely fast) compression per every block

Principles of Blosc

- Split data chunks in blocks internally (better cache utilization)
- Supports Shuffle and BitShuffle filters
- Uses parallelism at two levels:
  - Use multicores (multithreading)
  - Use SIMD in Intel/AMD processors (SSE2, AVX2) and ARM (NEON)

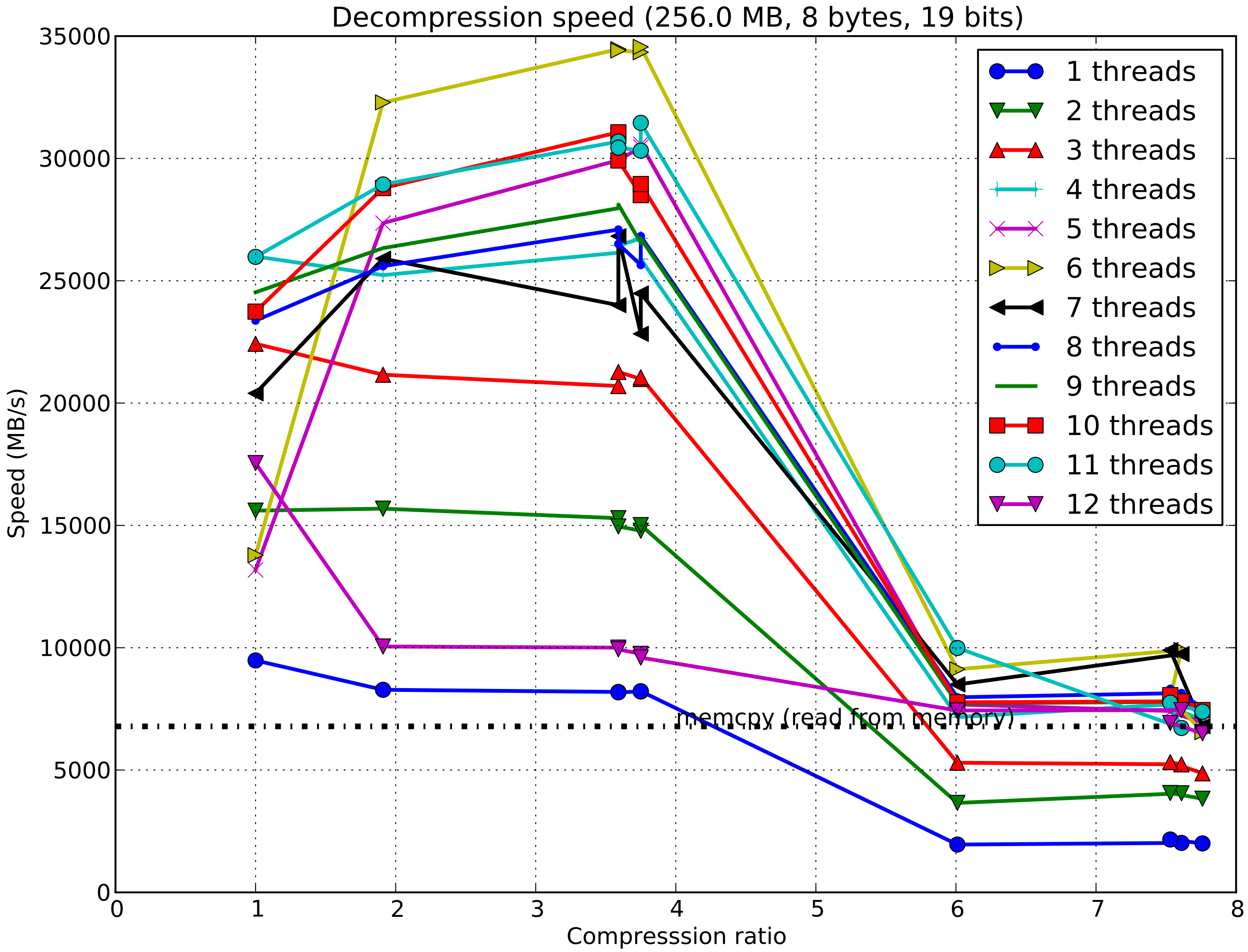
The Shuffle filter



- Shuffle works at byte level, and works well for integers or floats that vary smoothly

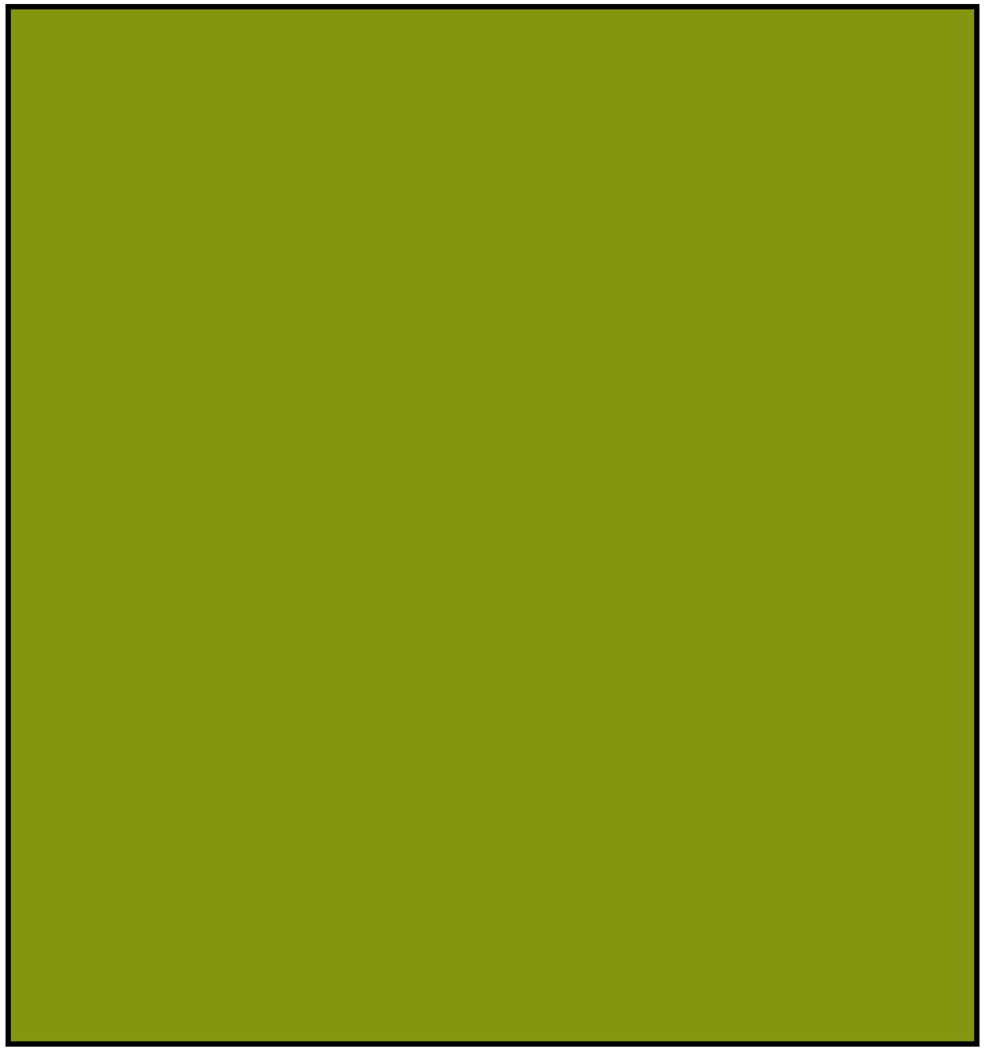
- There is also support for a BitShuffle filter that works at bit level

Blosc: Compressing Faster Than *memcpy()*



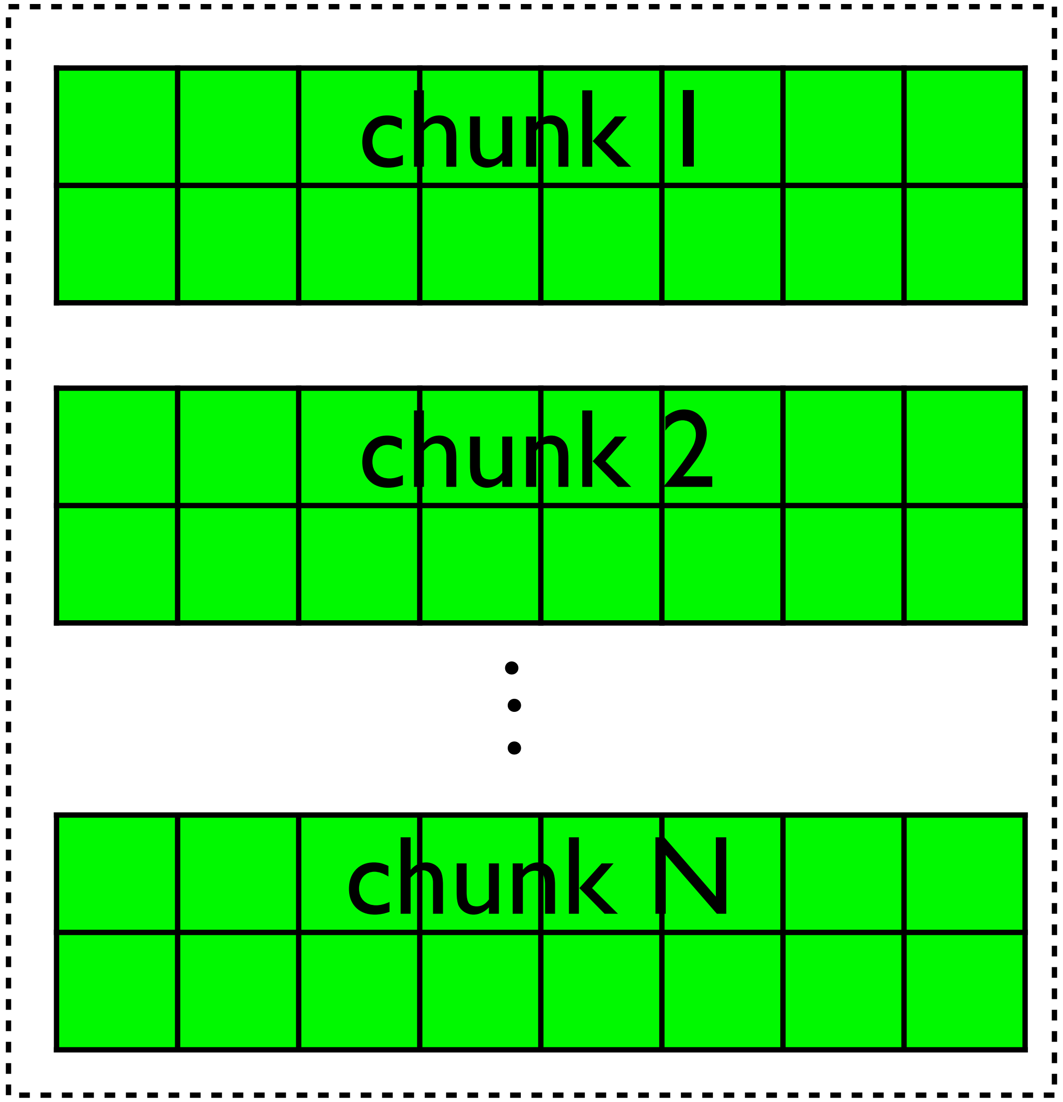
bcolz: a Data Container that Leverages the Blocking Technique

NumPy container



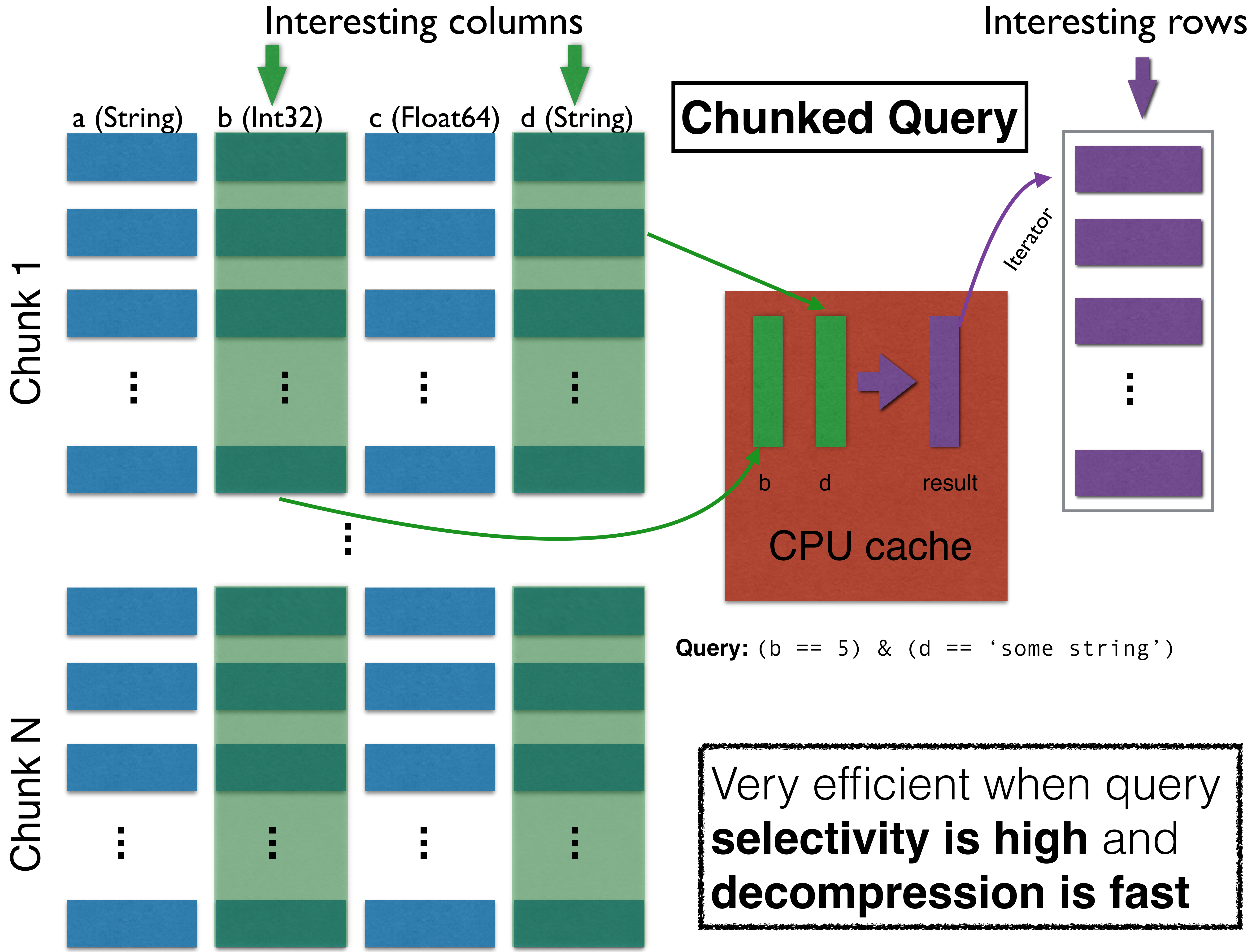
Contiguous memory

bcolz container



Discontiguous memory

Chunked Query

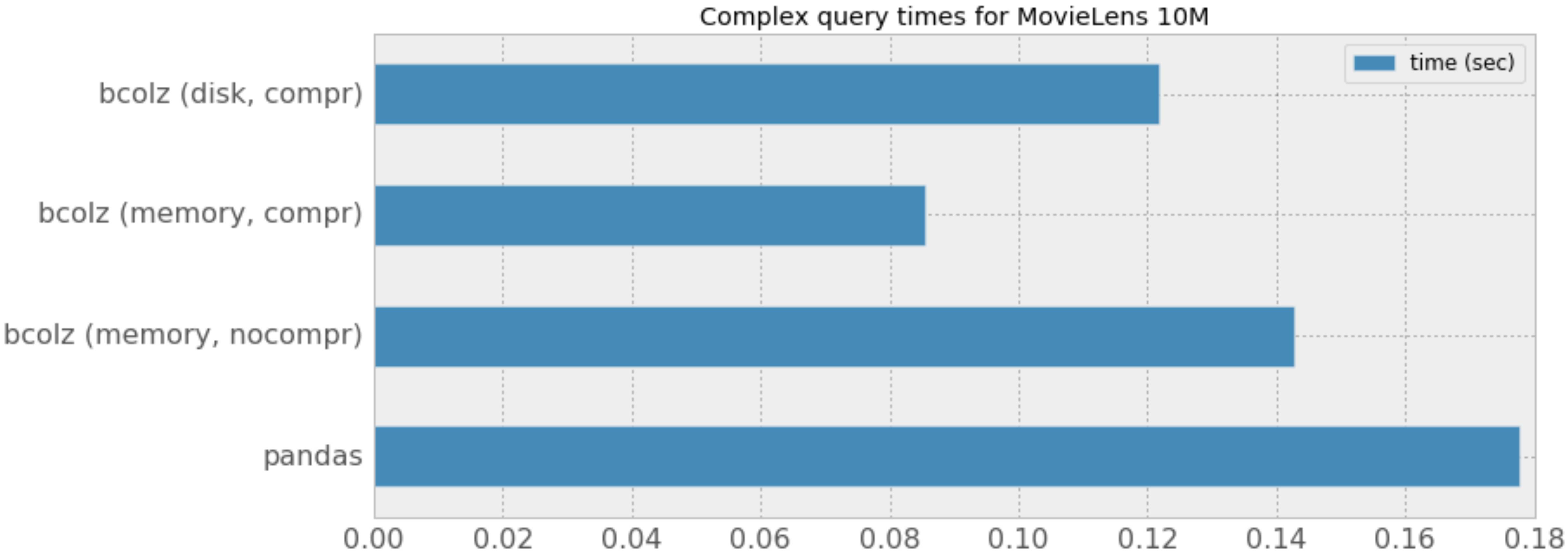


Query: `(b == 5) & (d == ‘some string’)`

Very efficient when query **selectivity is high** and **decompression is fast**

Query Times in `bcolz`

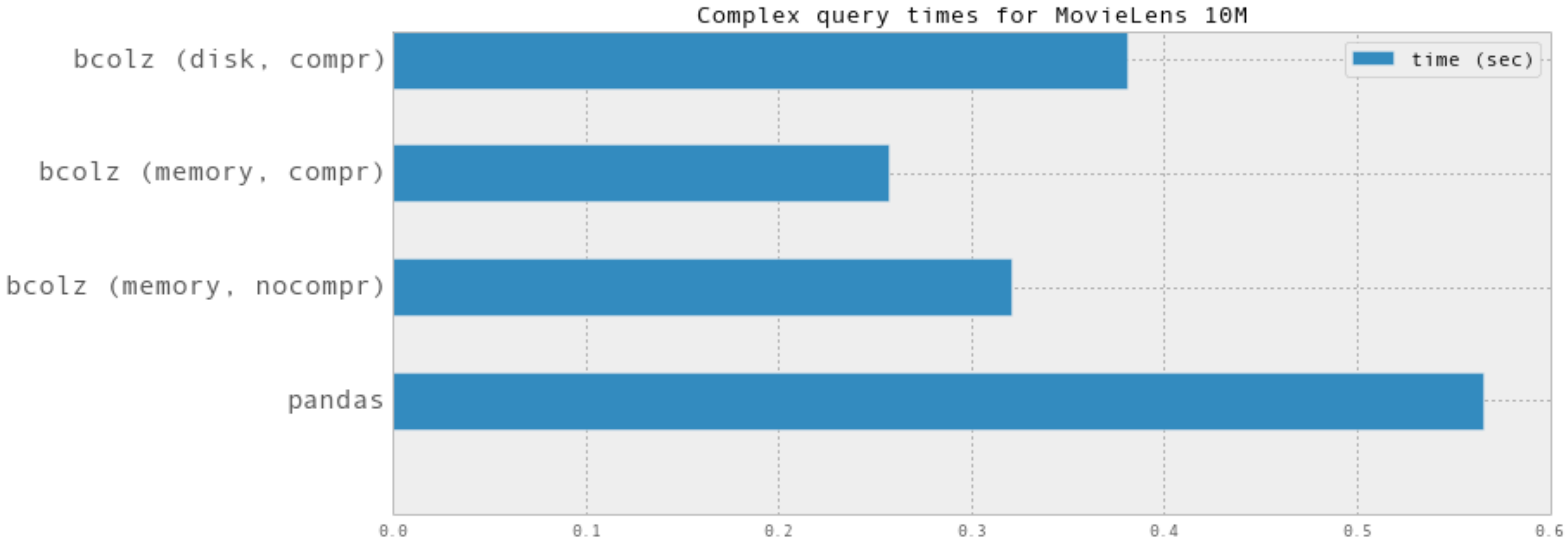
Recent server (Intel Xeon Skylake, 4 cores)
Compression **speeds** things up



Reference: https://github.com/Blosc/movielens-bench/blob/master/querying-ep14.ipynb

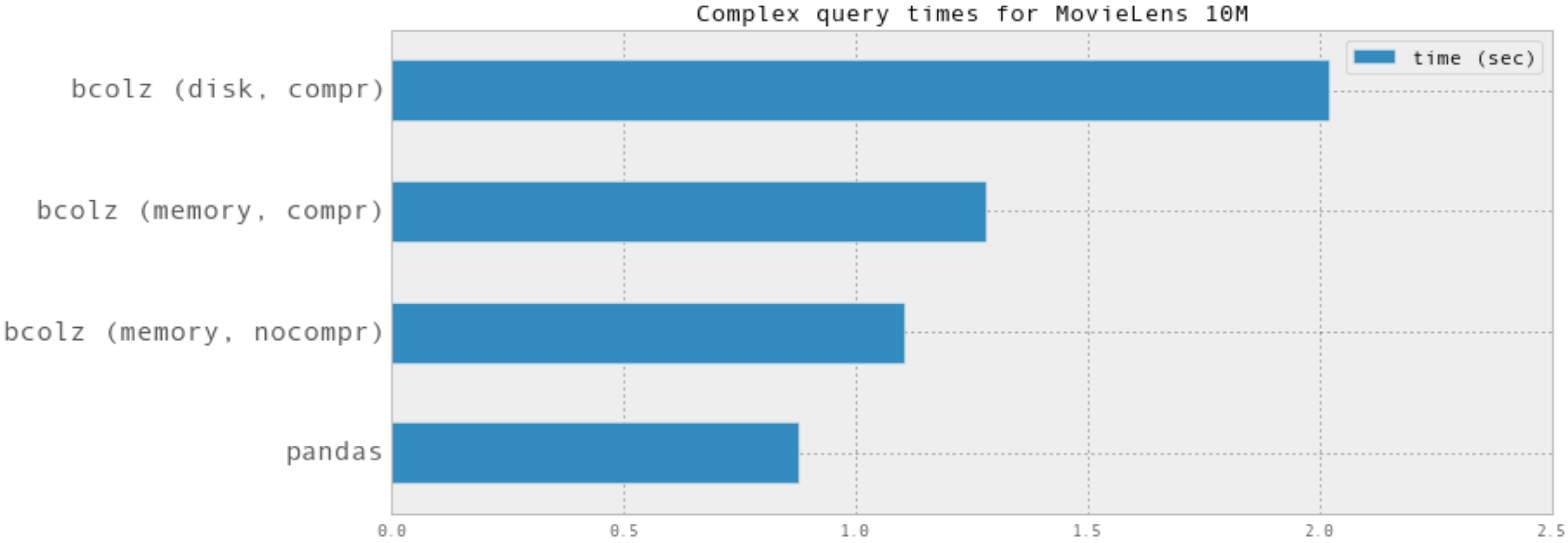
Query Times in `bcolz`

4-year old laptop (Intel Ivy-Bridge, 2 cores)
Compression still **speeds** things up

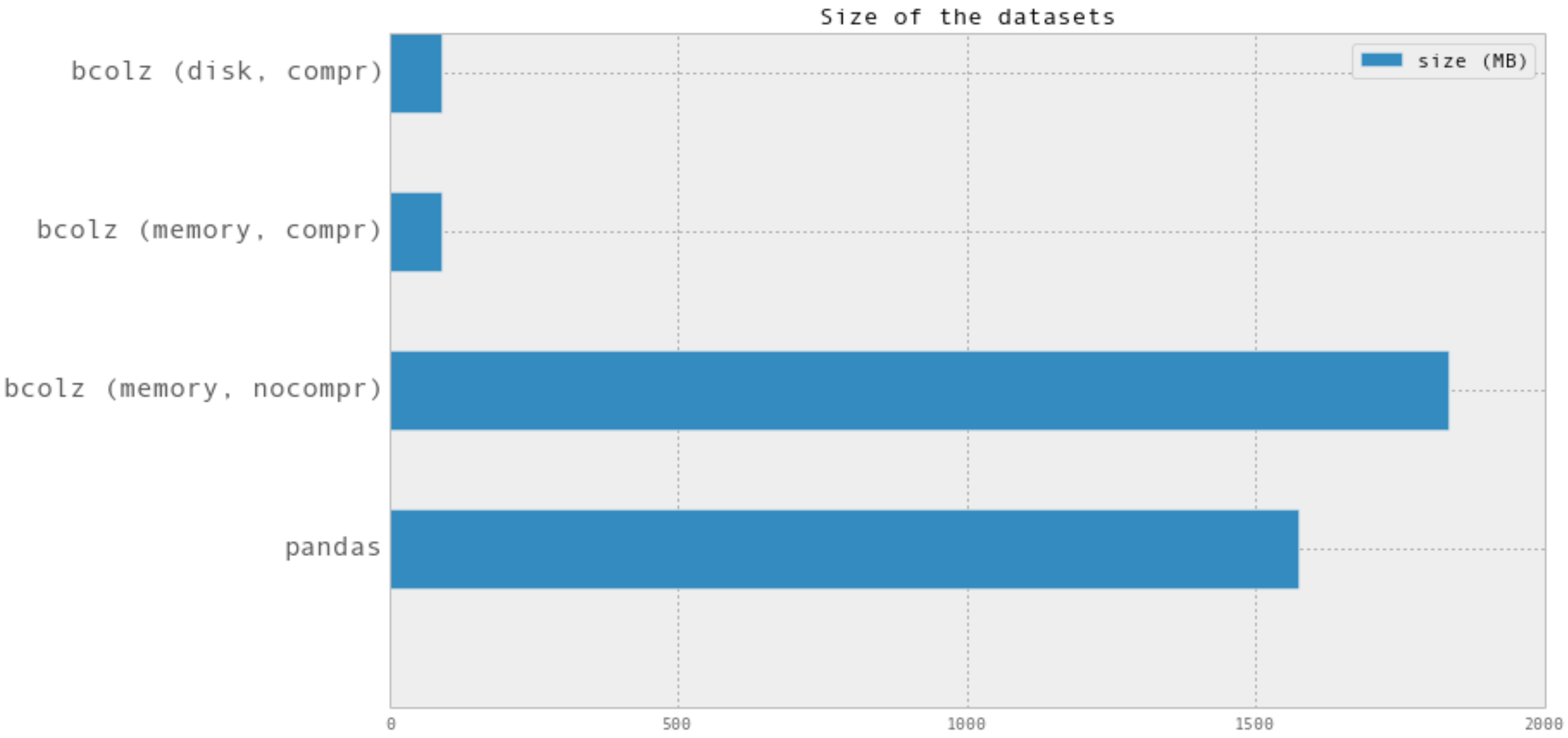


Query Times in `bcolz`

2010 laptop (Intel Core2, 2 cores)
Compression **slows** things down

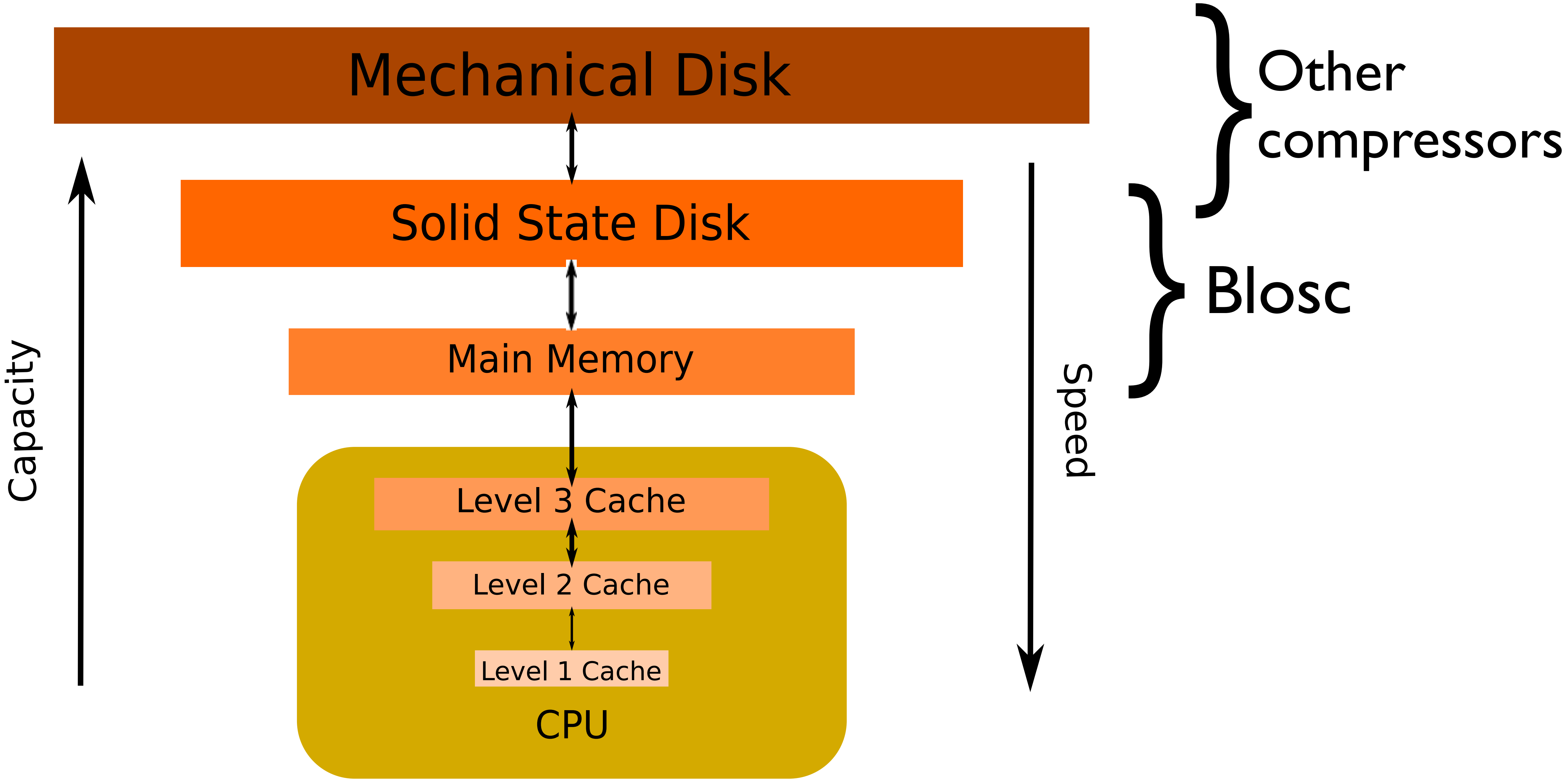


Sizes in `bcolz`



Do not forget compression main strength:
We can store more data while using same resources

Accelerating I/O With Blosc



“Blosc compressors are the fastest ones out there at this point; there is no better publicly available option that I'm aware of. That's not just ‘yet another compressor library’ case.”

— Ivan Smirnov
(advocating for Blosc inclusion in h5py)

Compression matters!

Closing Notes

- Due to the evolution in computer architecture, compression can be effective for two reasons:
  - We can work with more data using the same resources.
  - We can reduce the overhead of compression to near zero, and even beyond than that!

Thanks!