Efficient Memory Management

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

Francesc Alted

Continuum Analytics

Advanced Scientific Programming in Python, Kiel (Germany)
September 6, 2012

CONTINUUM
ANALYTICS

Overview

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

Motivation

Computing a Polynomial

We want to compute the next polynomial:

$$\mathbf{0.25x^3 + 0.75x^2 + 1.5x - 2}$$

in the range [-1, 1] with a step size of $2*10^{-7}$ in the x axis

...and want to do that as FAST as possible...

Using NumPy

```
import numpy as np

N = 10*1000*1000

x = np.linspace(-1, 1, N)

y = .25*x**3 + .75*x**2 - 1.5*x - 2
```

That takes around 1.60 sec on some machine (Intel Xeon E5520 @ 2.3 GHz). How to make it faster?

'Quick & Dirty' Approach: Parallelize

- Computing a polynomial is “embarrassingly” parallelizable: just divide the domain to compute in N chunks and evaluate the expression for each chunk.
- This can be easily implemented in Python by using the multiprocessing module (so as to bypass the GIL). See `poly-mp.py` script.
- Using 2 cores, the 1.60 sec is reduced down to 1.18 sec, which is a 1.35x improvement. Not bad.
- We are done! Or perhaps not?

A Better Approach: Optimize

The NumPy expression:

(I) $\mathbf{0.25x^3 + 0.75x^2 + 1.5x - 2}$

can be rewritten as:

(II) $\mathbf{((0.25x + 0.75)x + 1.5)x - 2}$

- Exec time goes from 1.60 sec to 0.30 sec
- Much faster (4x) than using two processors with the multiprocessing approach (1.18 sec).

First Lesson To Be Learned

- Do not blindly try to parallelize right away:

  Optimizing normally gives better results

  And a serial codebase is normally much easier to code and debug!

Use numexpr

Numexpr is a JIT compiler, based on NumPy, that optimizes the evaluation of complex expressions. Usage is simple:

```
import numpy as np
import numexpr as ne
N = 10*1000*1000
x = np.linspace(-1, 1, N)
y = ne.evaluate(“.25*x**3 + .75*x**2 - 1.5*x - 2”)
```

This takes 0.14 sec to complete (11x faster than the original NumPy: 1.60 sec)

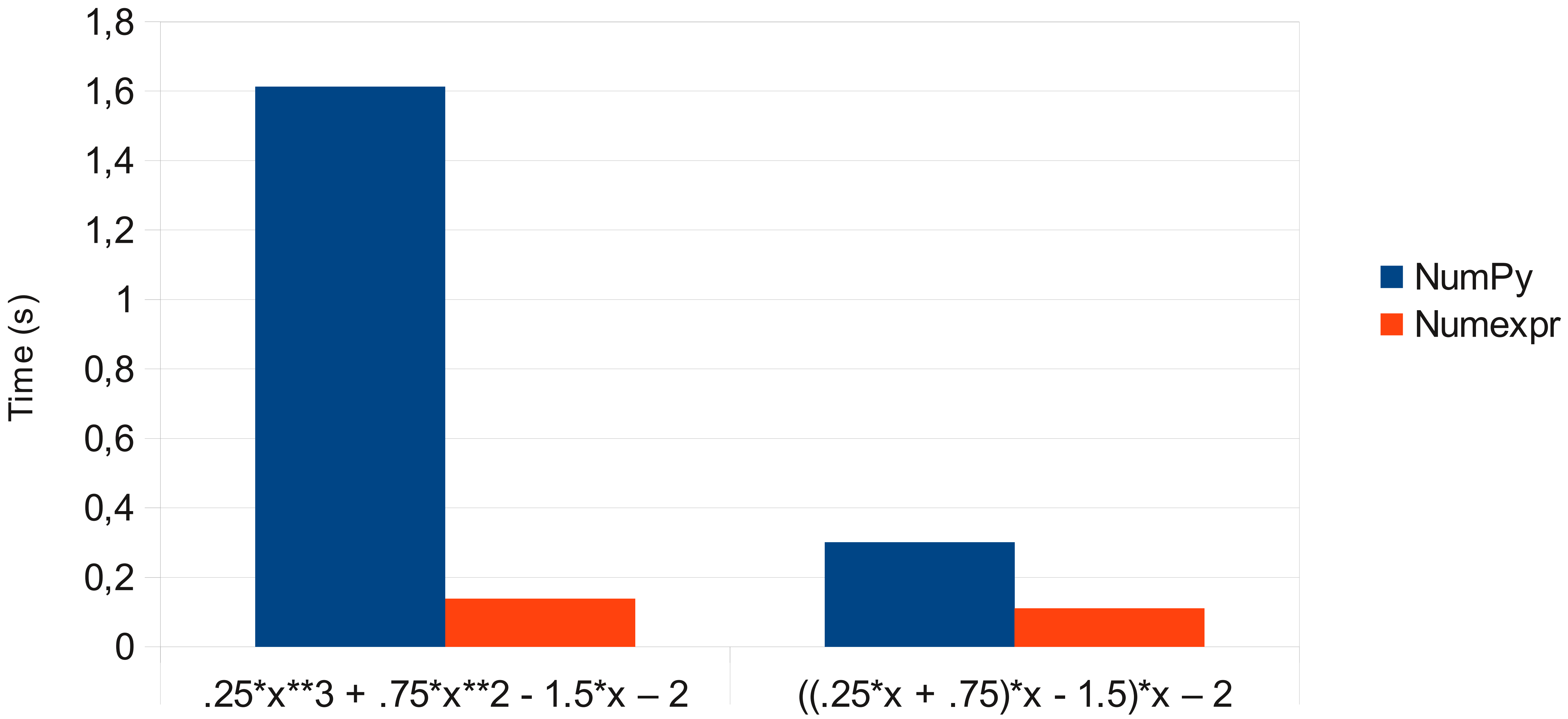
Fine Tuning numexpr

Numexpr is also sensible to computer-friendly expressions like:

(II) **((0.25x + 0.75)x + 1.5)x - 2**

- This takes 0.11 sec (3x faster than NumPy)
- 0.14 sec were needed for the original expression, that's a 25% faster

Time to evaluate polynomial (1 thread)



Power Expansion

Numexpr expands expression:

```
0.25*x**3 + 0.75*x**2 + 1.5*x - 2
```

to:

```
0.25*x*x*x + 0.75*x*x + 1.5*x*x - 2
```

so, no need to use the expensive pow()

One Remaining Question

Why numexpr can execute this expression:

((0.25x + 0.75)x + 1.5)x - 2

3x faster, even using a single core?

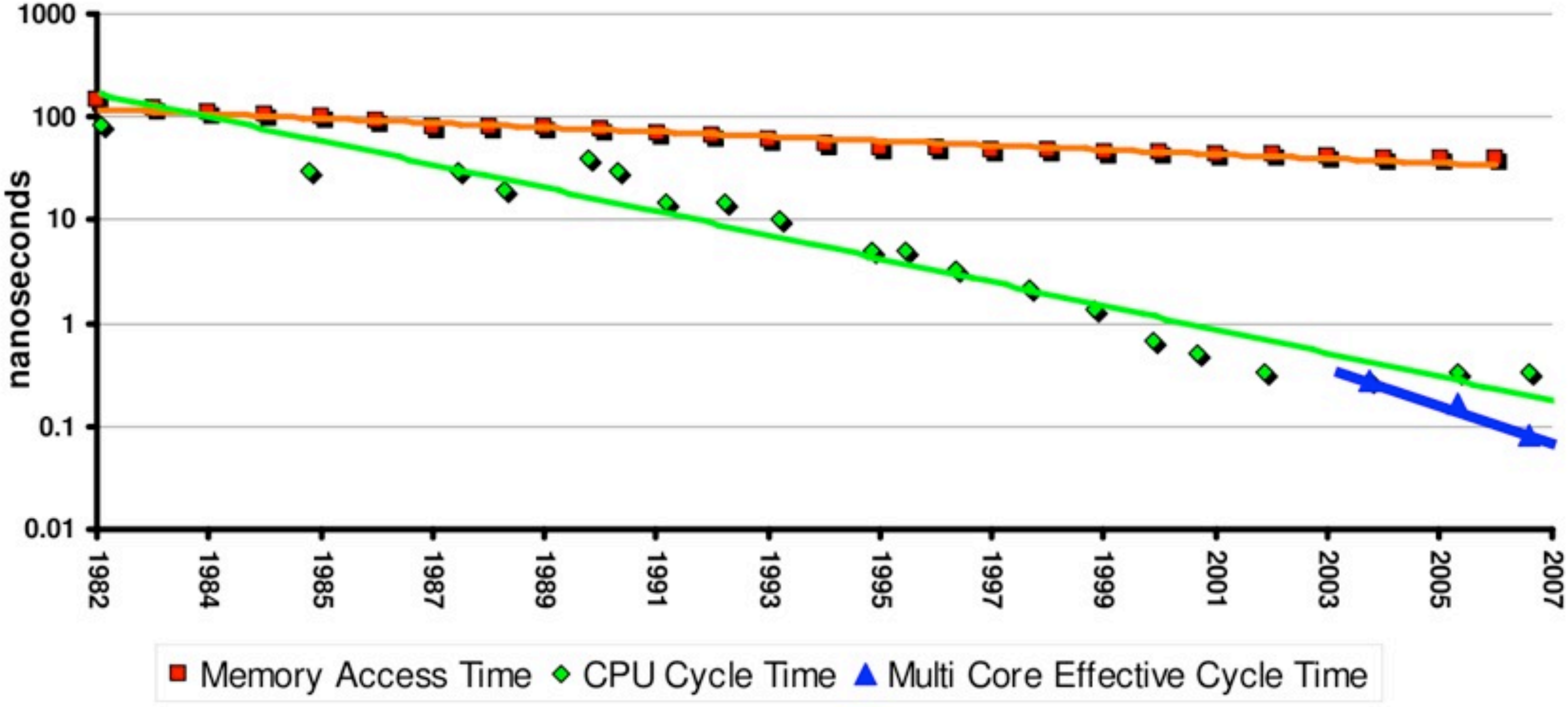
Short answer: making a more efficient use of the memory resource

The Starving CPU Problem

The Starving CPU Problem

- Current CPUs typically stay bored, doing nothing most of the time
- Why so?
- Because they are 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.”

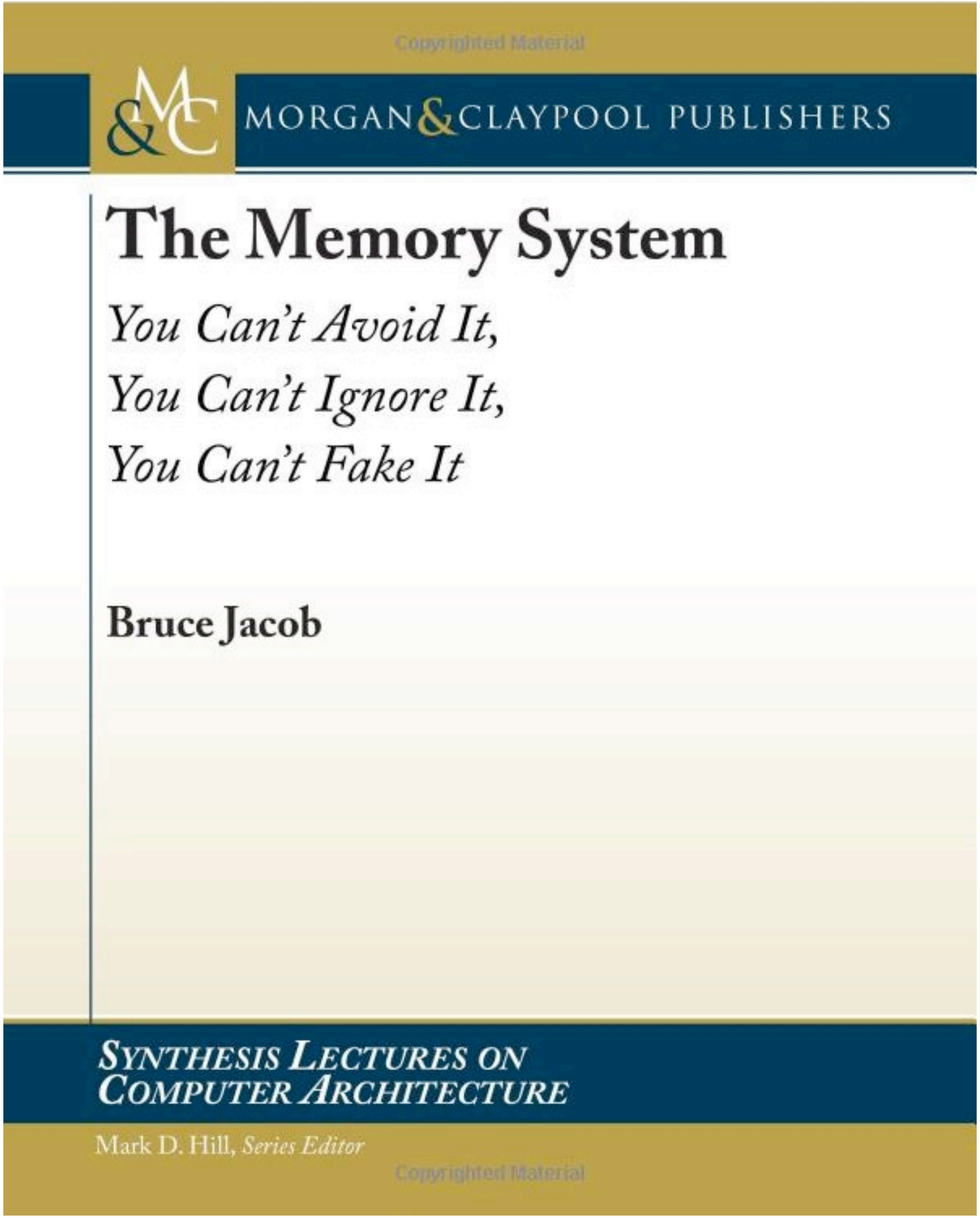
– Kevin Dowd, after his book “High Performance Computing”,
O’Reilly & Associates, Inc, 1993

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



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*

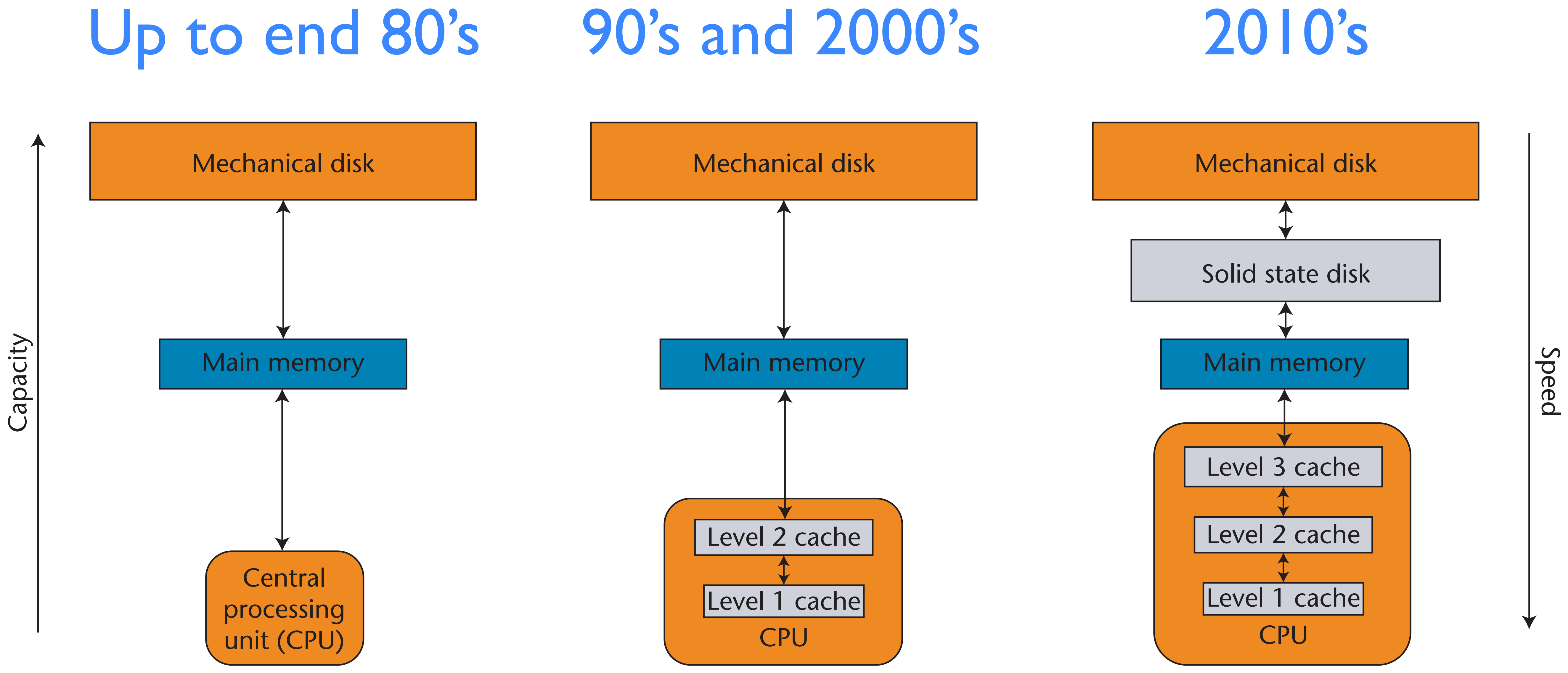
The Status of CPU Starvation in 2012

- Memory latency is much slower (between 250x and 500x) 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

CPU Cache Evolution



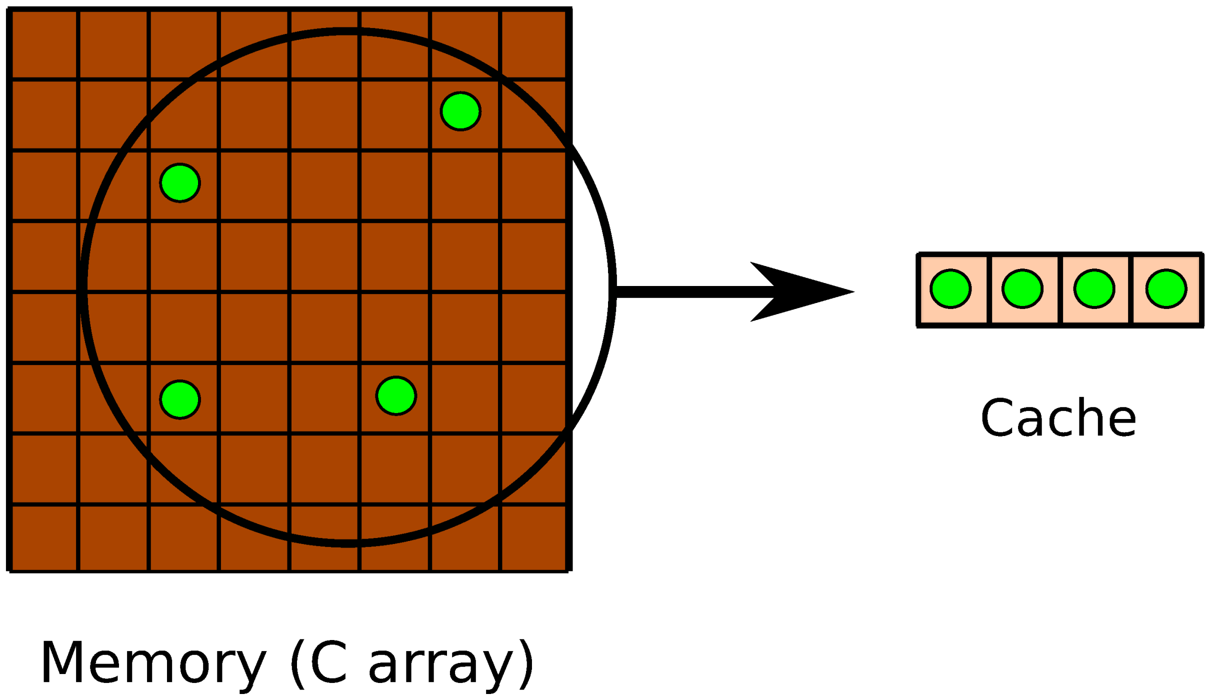
When CPU Caches Are Effective?

Mainly in a couple of scenarios:

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

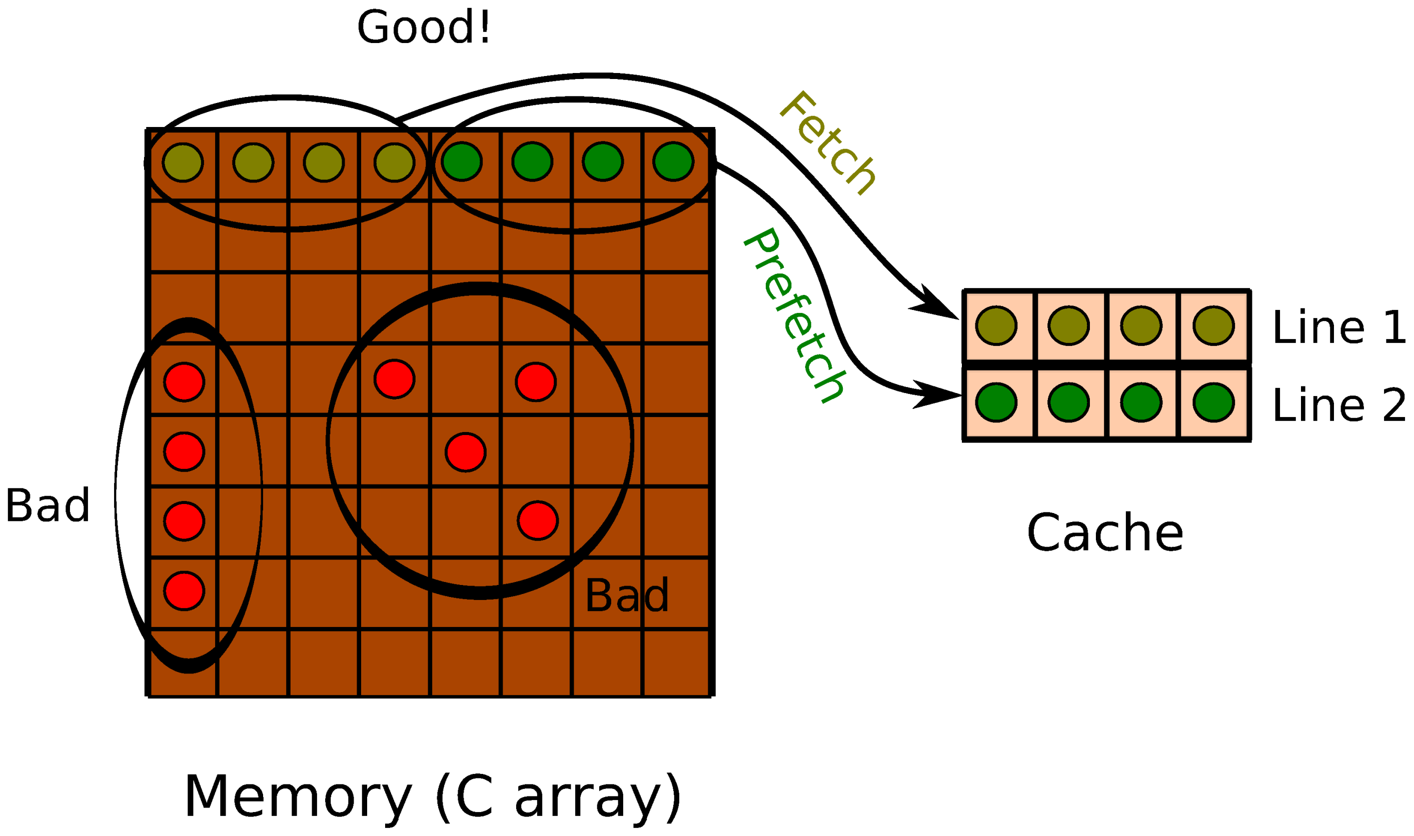
Time Locality

Parts of the dataset are reused



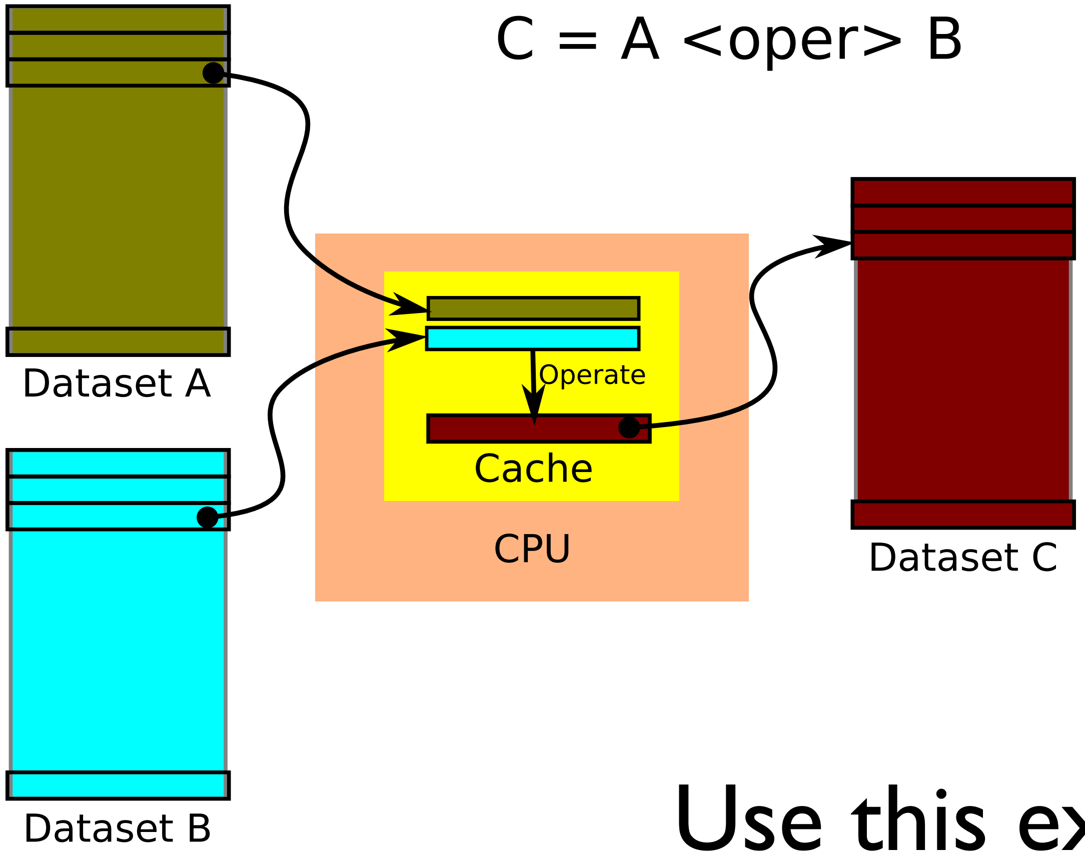
Spatial Locality

Dataset is accessed sequentially



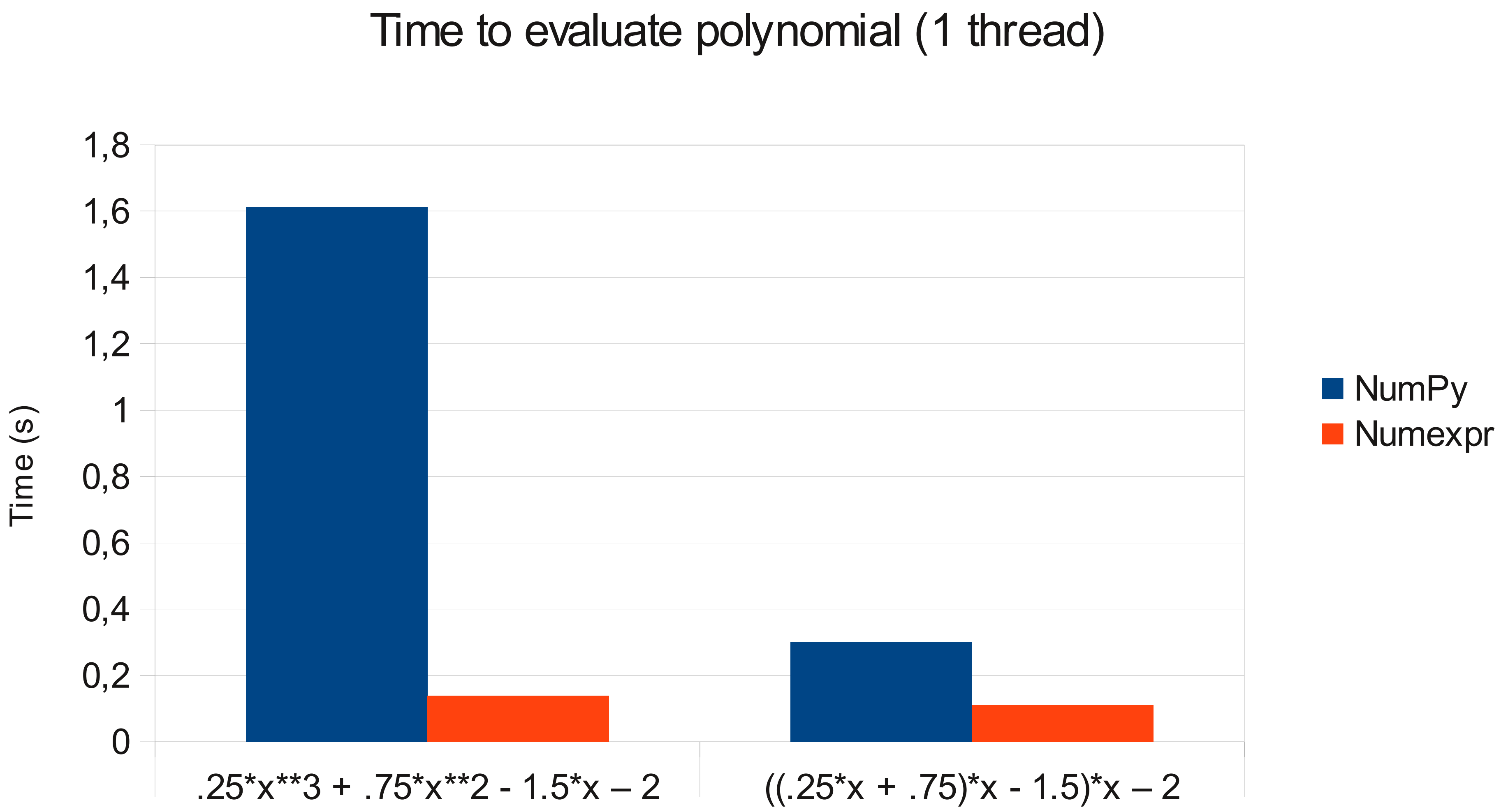
The Blocking Technique

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

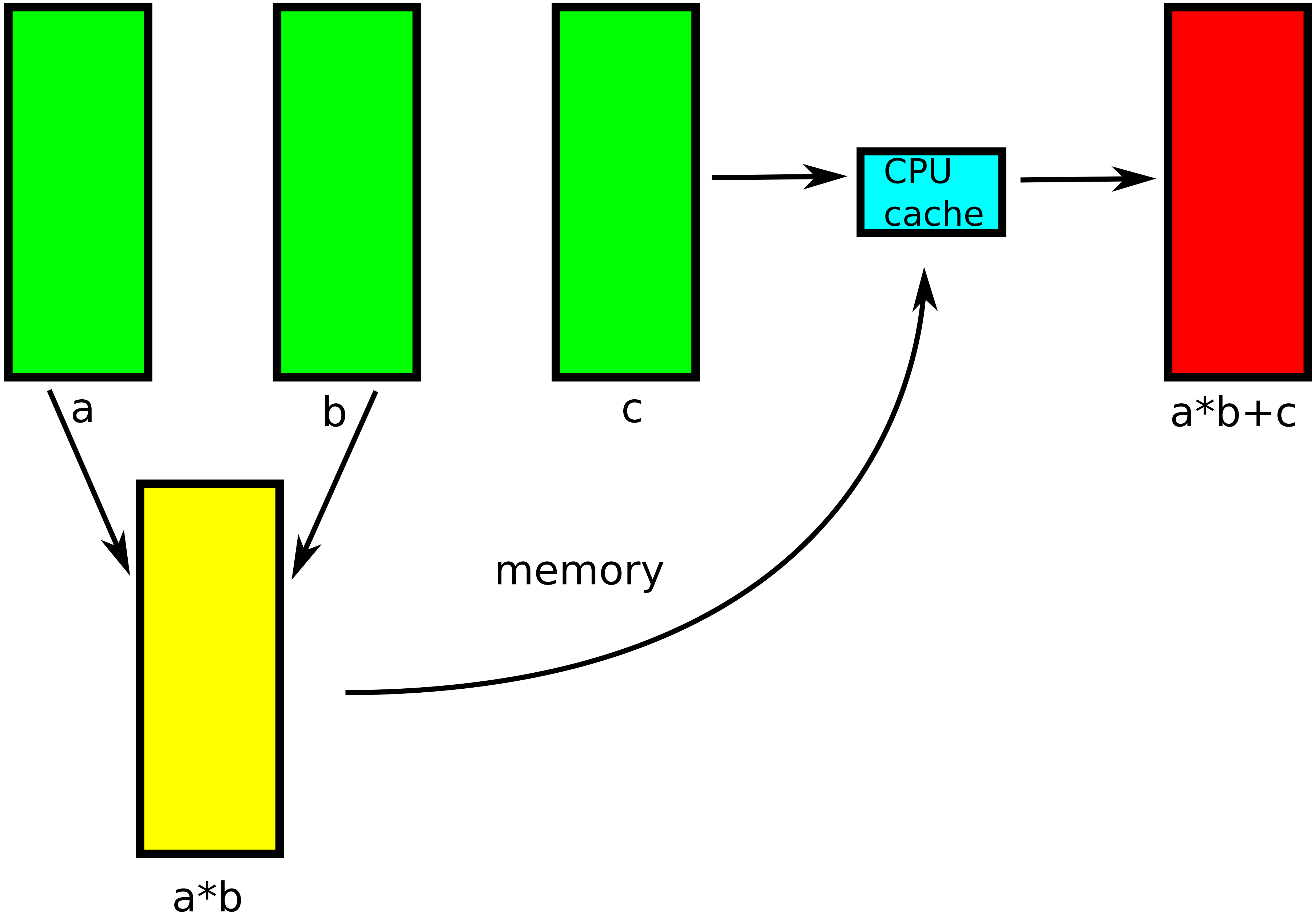


Use this extensively to leverage spatial and temporal localities

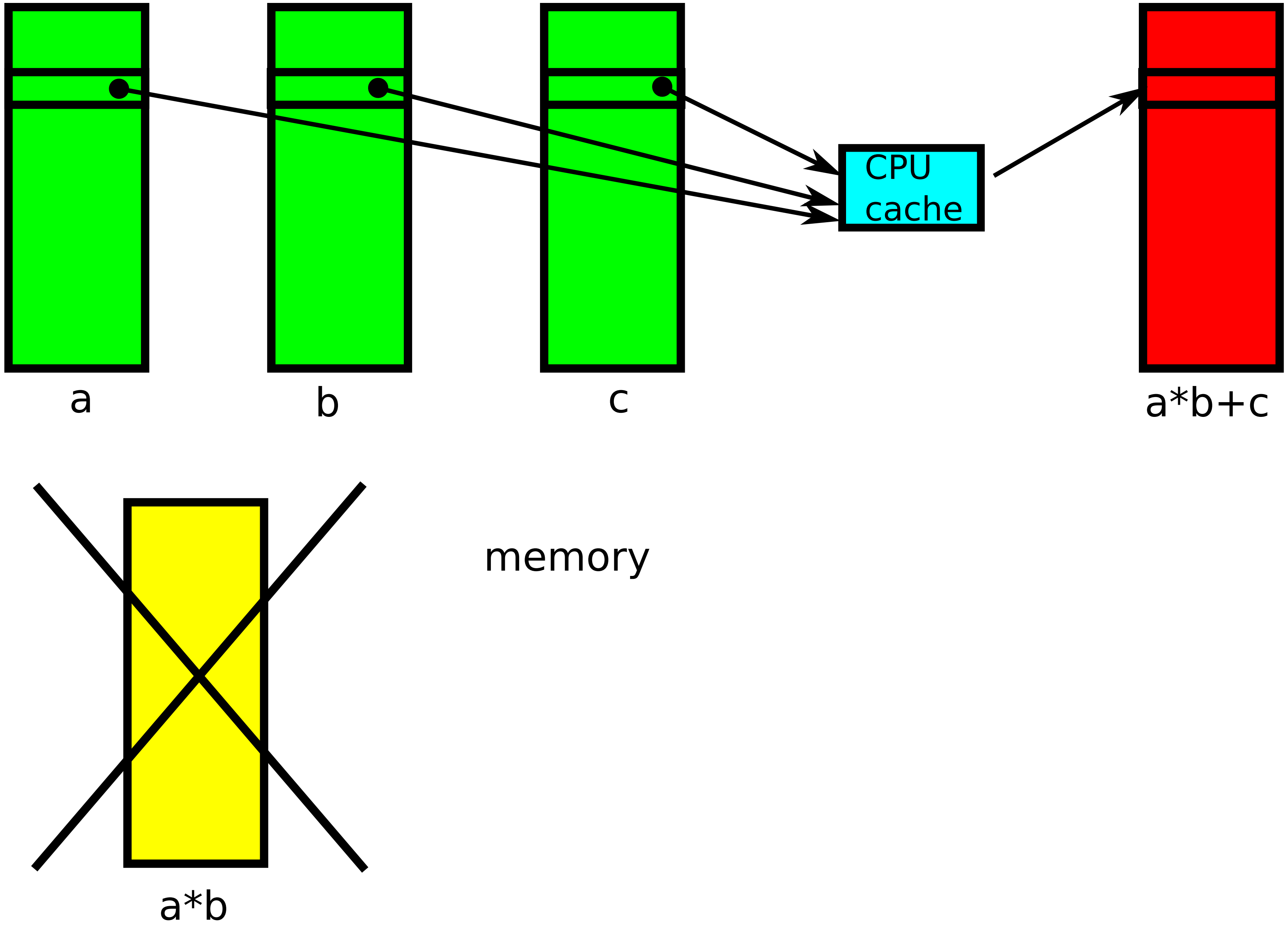
Time To Answer Pending Questions



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

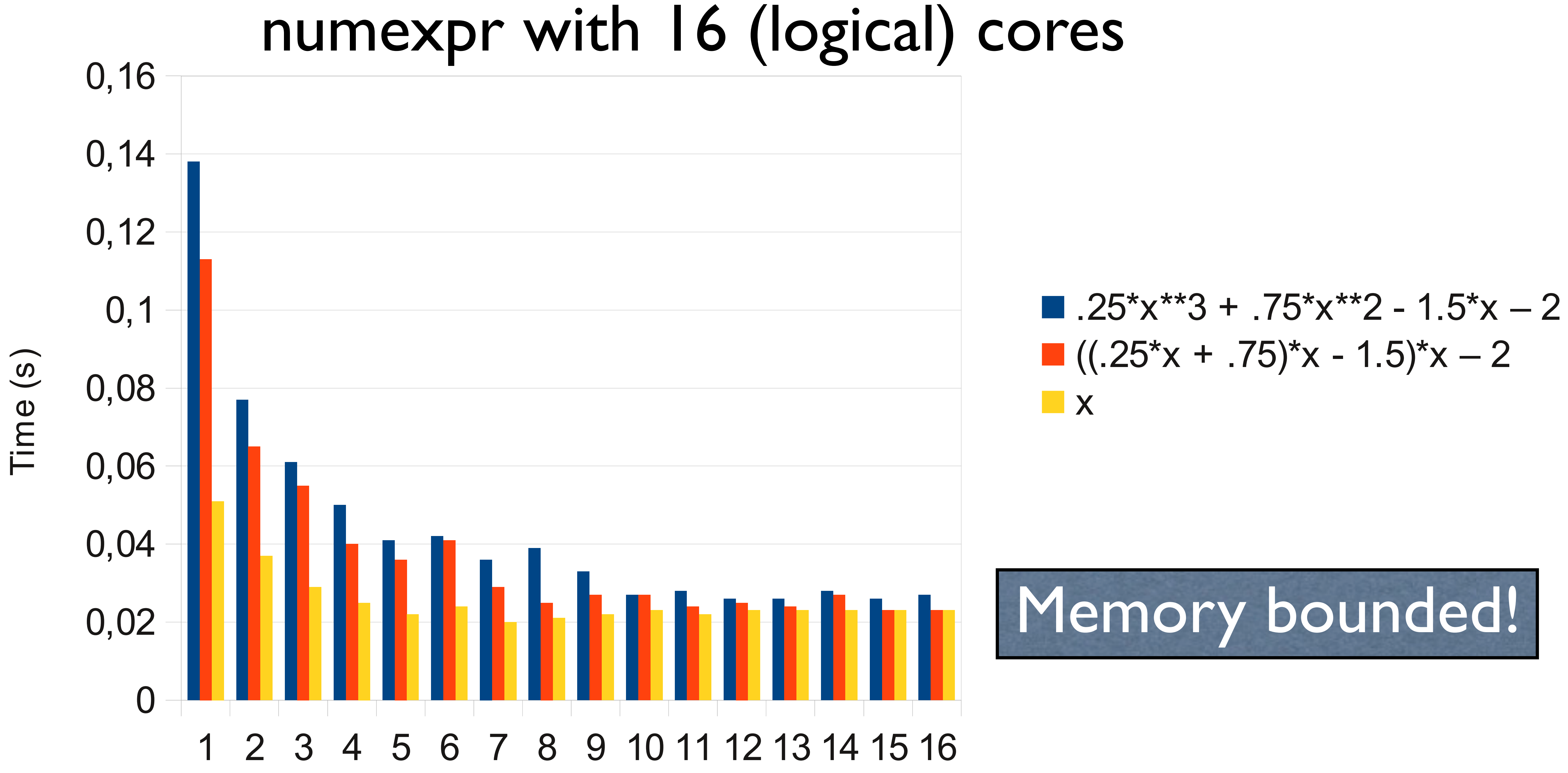


Computing "a*b+c" with Numexpr. Temporaries in memory are avoided.



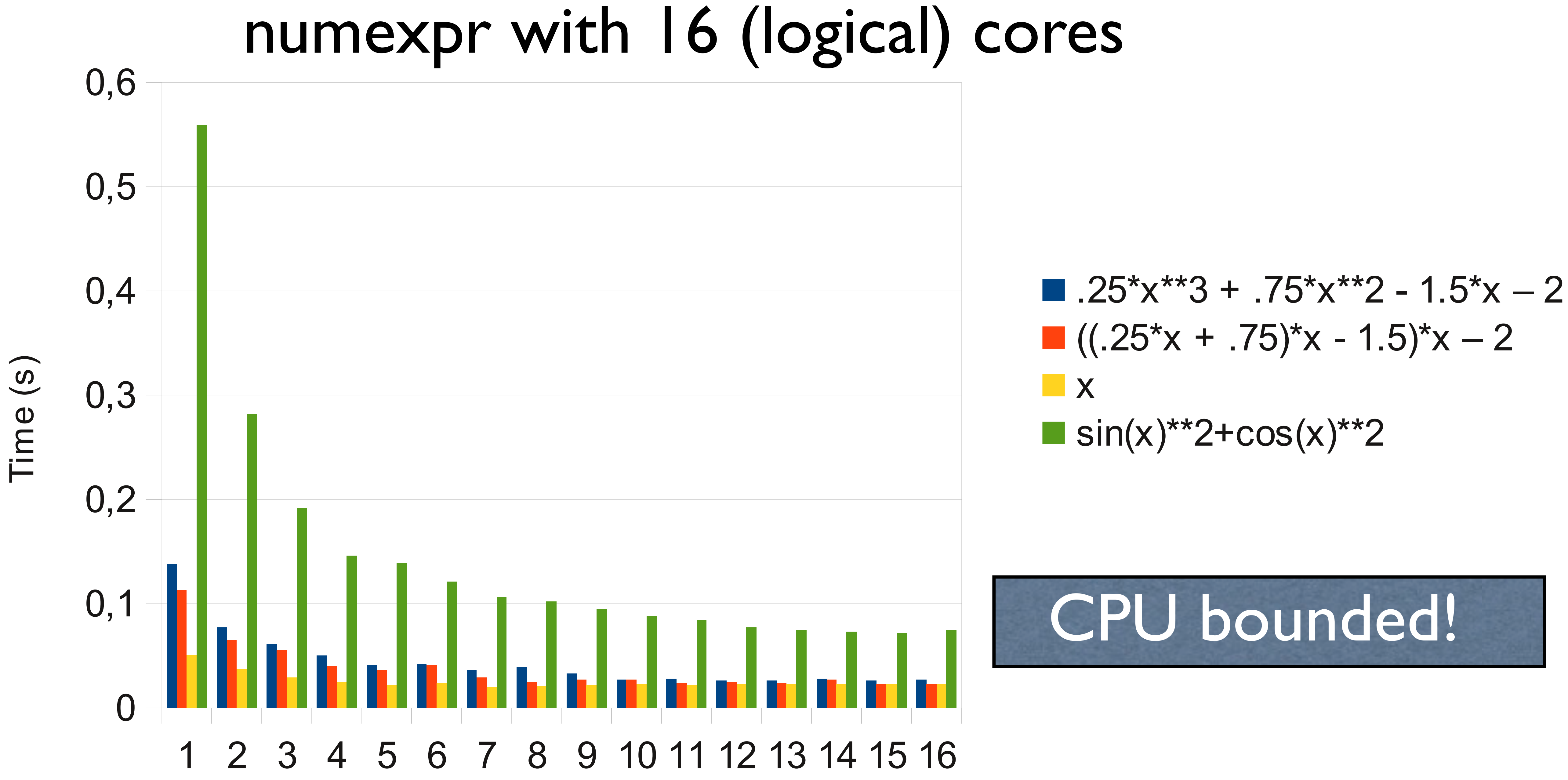
Multithreaded numexpr and Beyond: Numba

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]
```

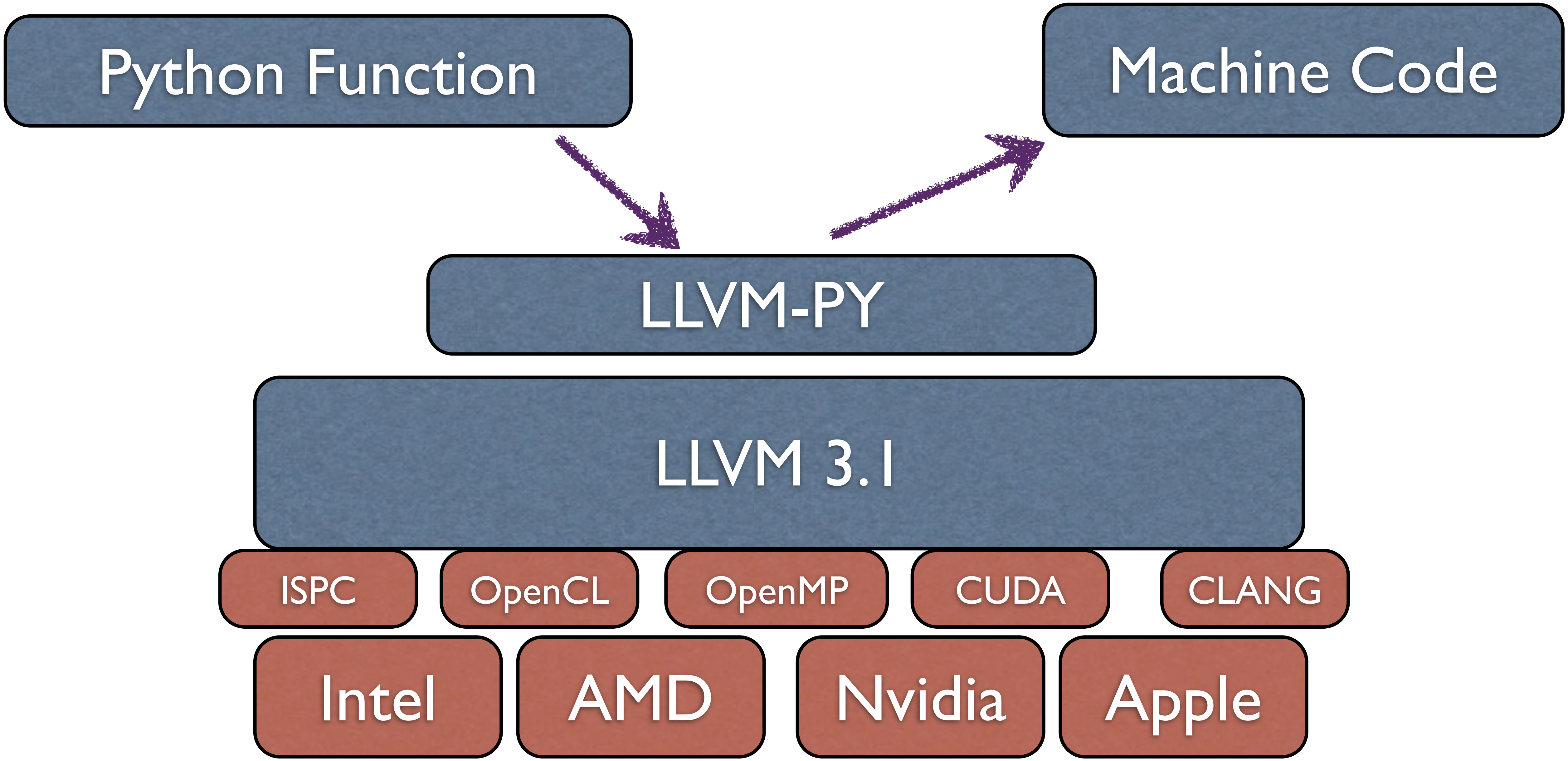
- In particular, it cannot deal with things like:

```
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



Numba Example: Computing the Polynomial

```
from numba import d
from numba.decorators import jit as jit
import numpy as np

N = 10*1000*1000

x = np.linspace(-1, 1, N)
y = np.empty(N, dtype=np.float64)

@jit(arg_types=[d[:], d[:]])
def poly(x, y):
    for i in range(N):
        # y[i] = 0.25*x[i]**3 + 0.75*x[i]**2 + 1.5*x[i] - 2
        y[i] = ((0.25*x[i] + 0.75)*x[i] + 1.5)*x[i] - 2

poly(x, y)  # run through Numba!
```

Times for Computing the Polynomial (In Seconds)

Poly version	(I)	(II)
Numpy	1.086	0.505
numexpr	0.108	0.096
Numba	0.055	0.054
Pure C, OpenMP	0.215	0.054

- Compilation time for Numba: 0.019 sec
- Run on Mac OSX, Core2 Duo @ 2.13 GHz

Second Lesson of the Day

- Before trying to optimize yourself:

  Be aware about existing libraries out there

  It is pretty difficult to beat performance professionals!

Optimal Containers for Big Data

The Need for a Good Data Container

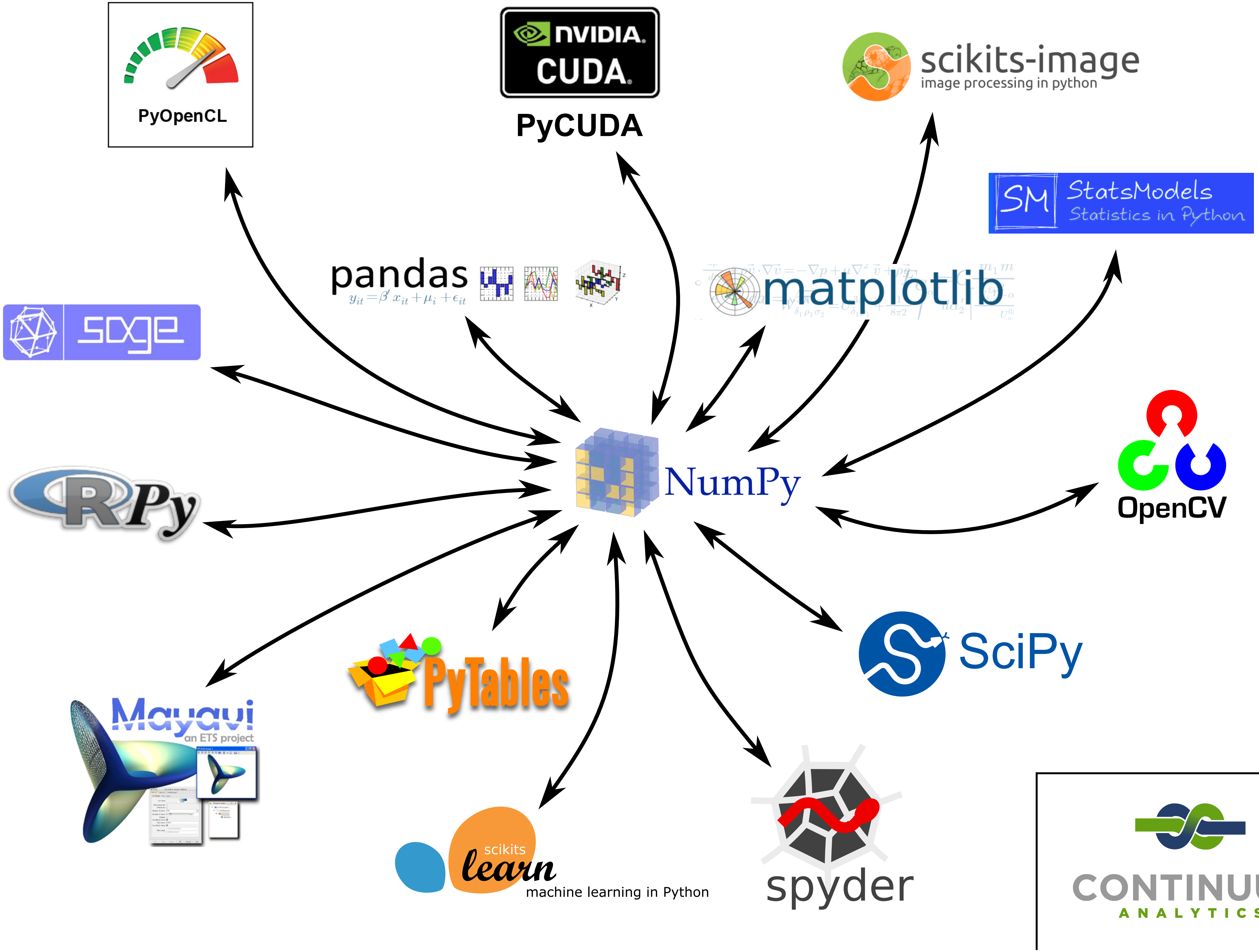
- Too many times we are focused on computing as fast as possible
- But we have seen how important data access is
- Hence, having an optimal data structure is critical for getting good performance when processing very large datasets

NumPy: A De Facto Data Container

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

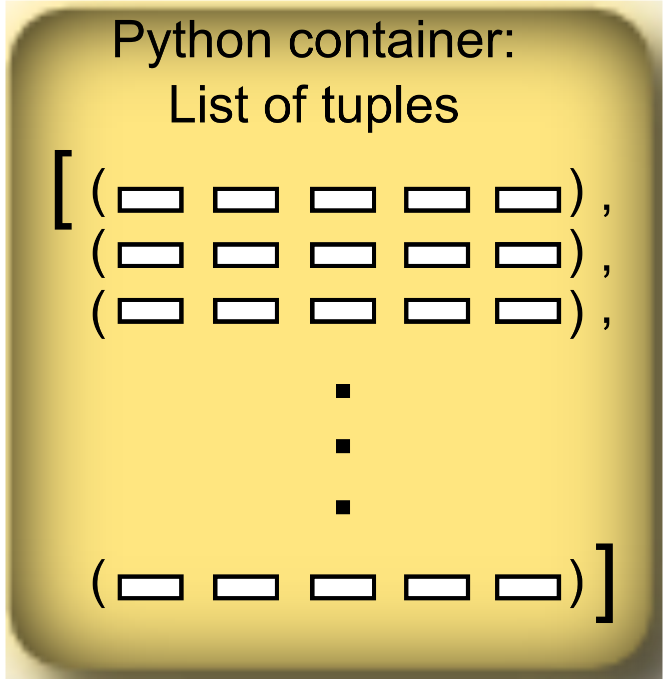
NumPy Advantages

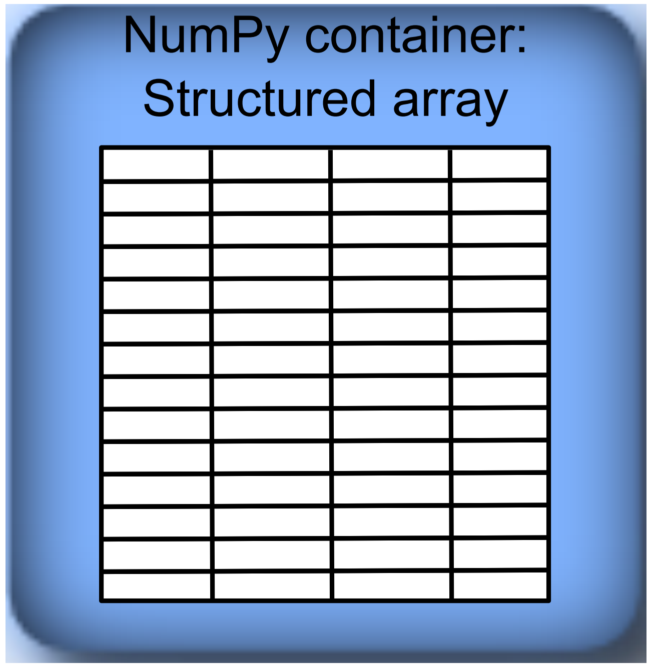
- Multidimensional data container
- Efficient data access
- Powerful weaponry for data handling
- Efficient in-memory storage



CONTINUUM
ANALYTICS

NumPy As an Efficient Data Container





Faster creation time

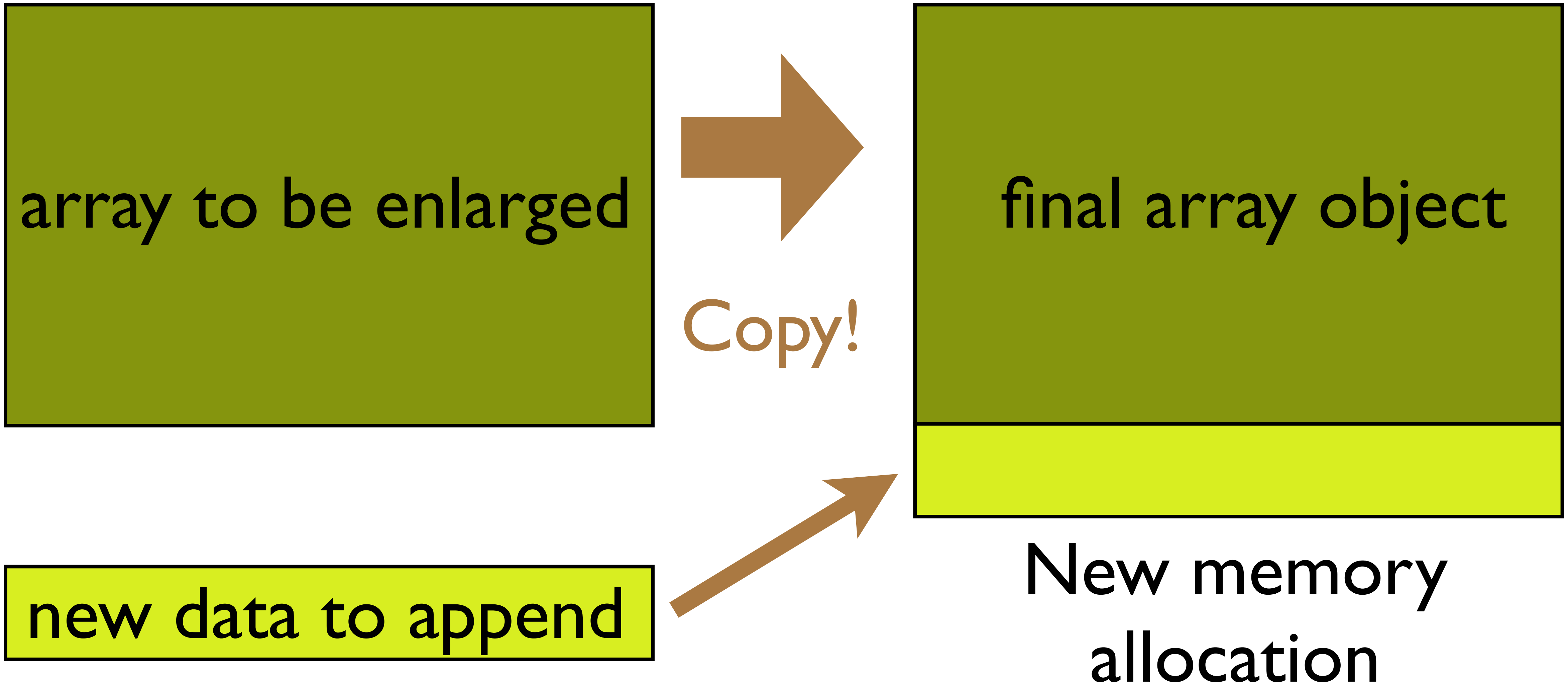
No fragmentation

One data byte ~ one in-memory byte

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

Appending Data in Large NumPy Objects

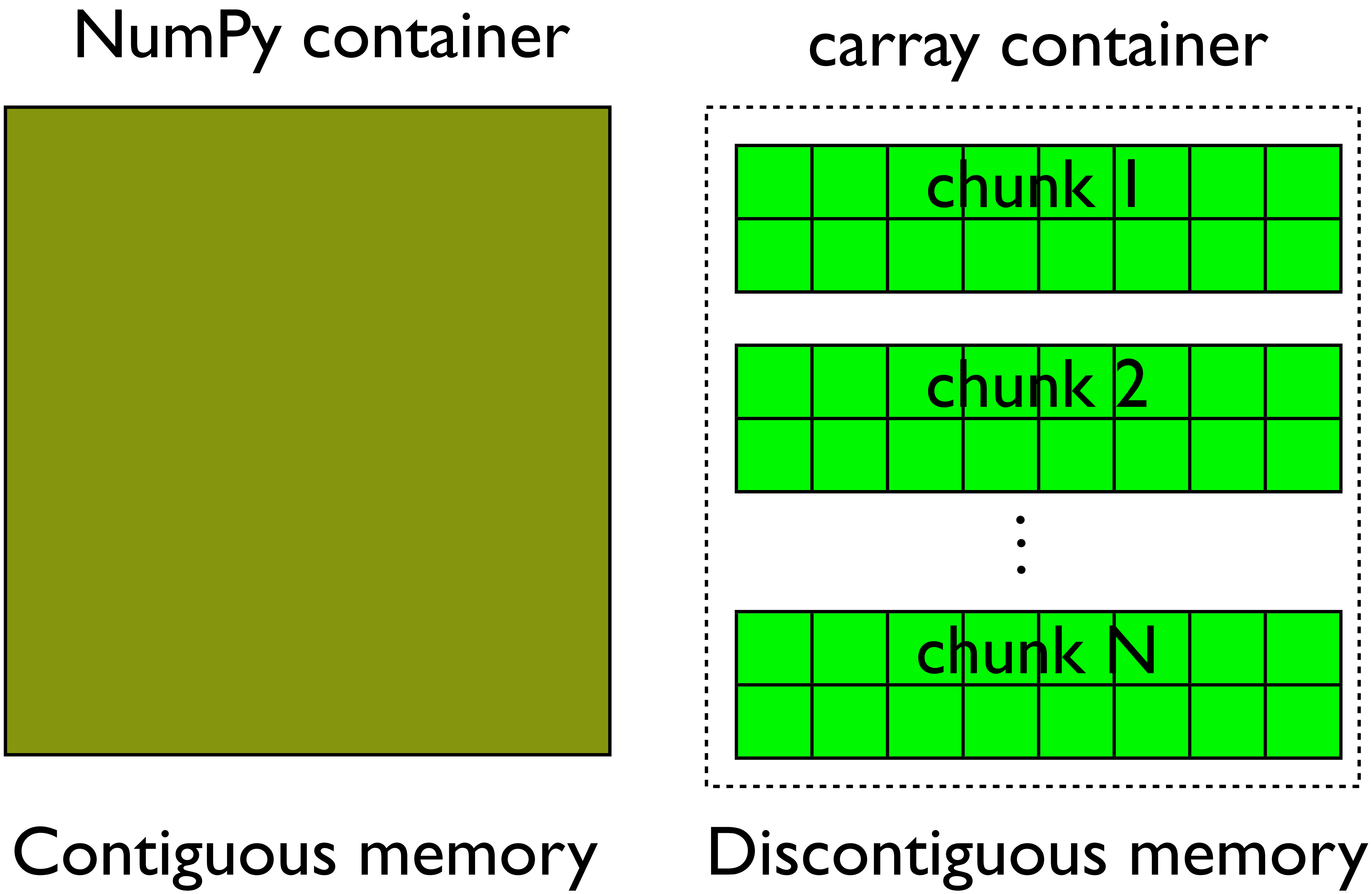


- Normally a `realloc()` syscall will not succeed
- Both memory areas have to exist **simultaneously**

carray

- carray is a data container that can be used in a similar way than the one in NumPy
- The main difference is that data storage is **chunked**, not **contiguous**
- Containers can be enlarged without **copying** the original container

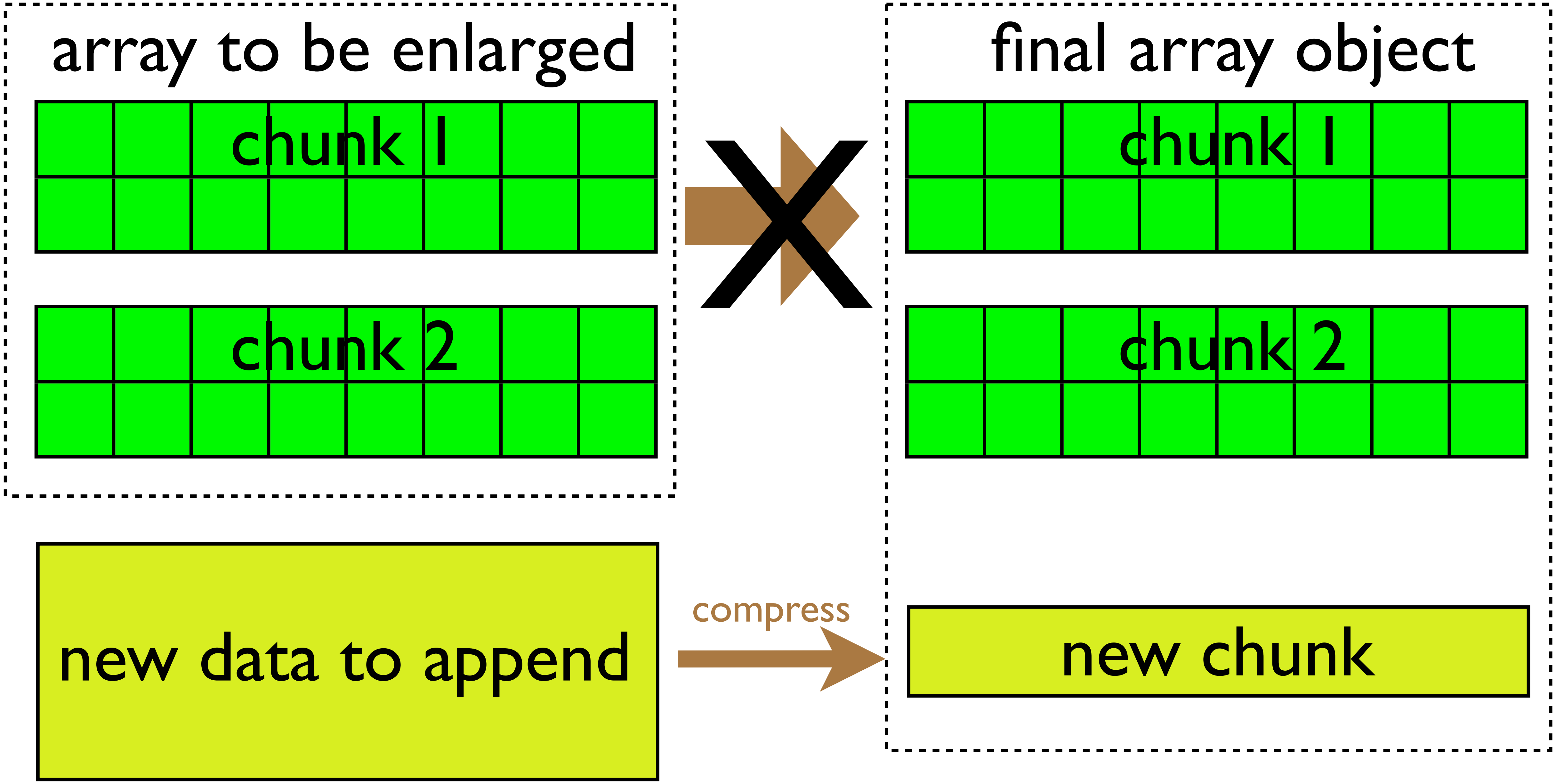
Contiguous vs Chunked



Why Chunking?

- Chunking means more difficulty handling data, so why bother?
  - Efficient enlarging and shrinking
  - Compression is possible

Appending data in carray



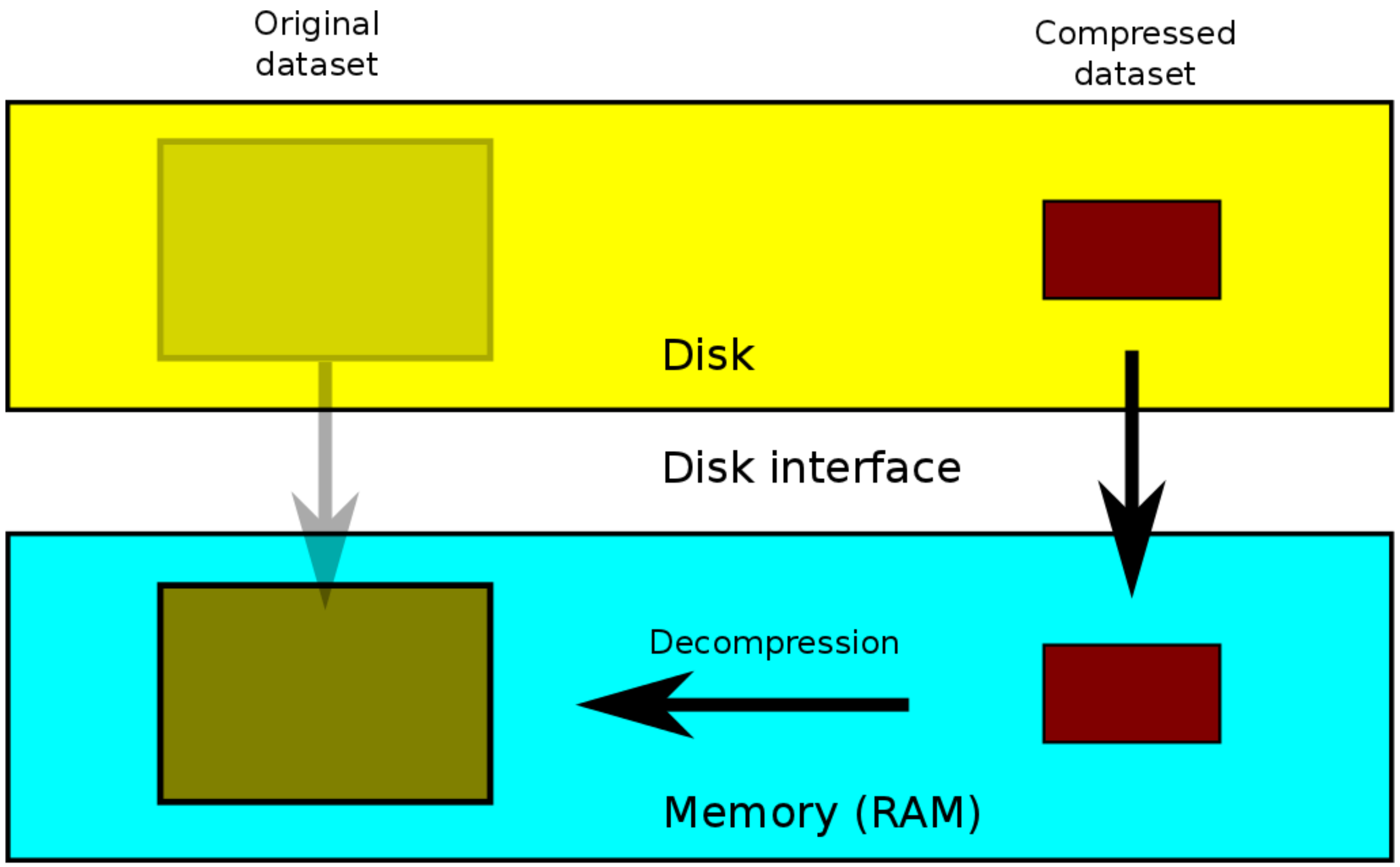
Only a small amount of data has to be **compressed**

Why Compression?

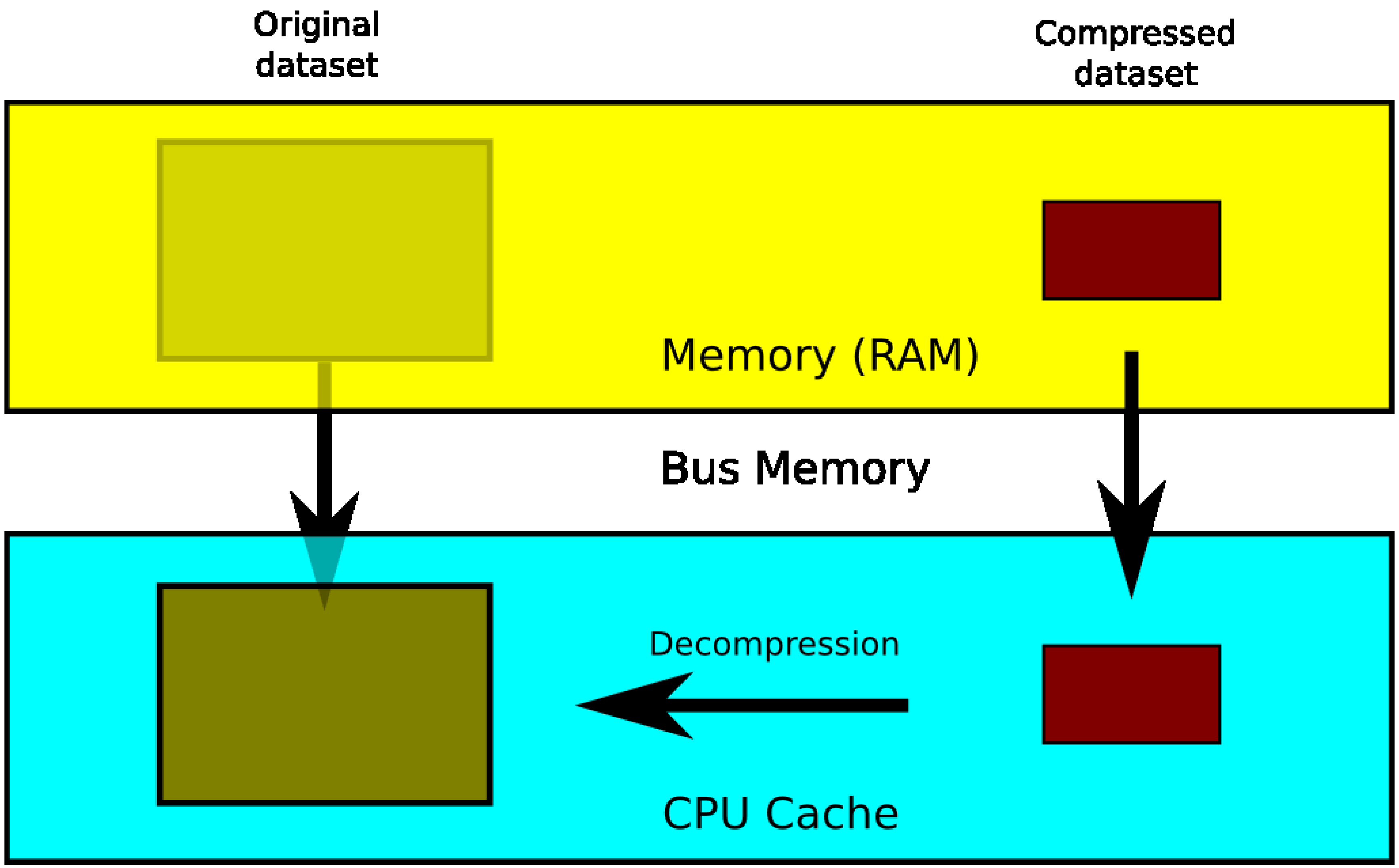
Lets you store more data using the same space

Uses more CPU, but CPU time is cheap

Overall, it can make I/O faster

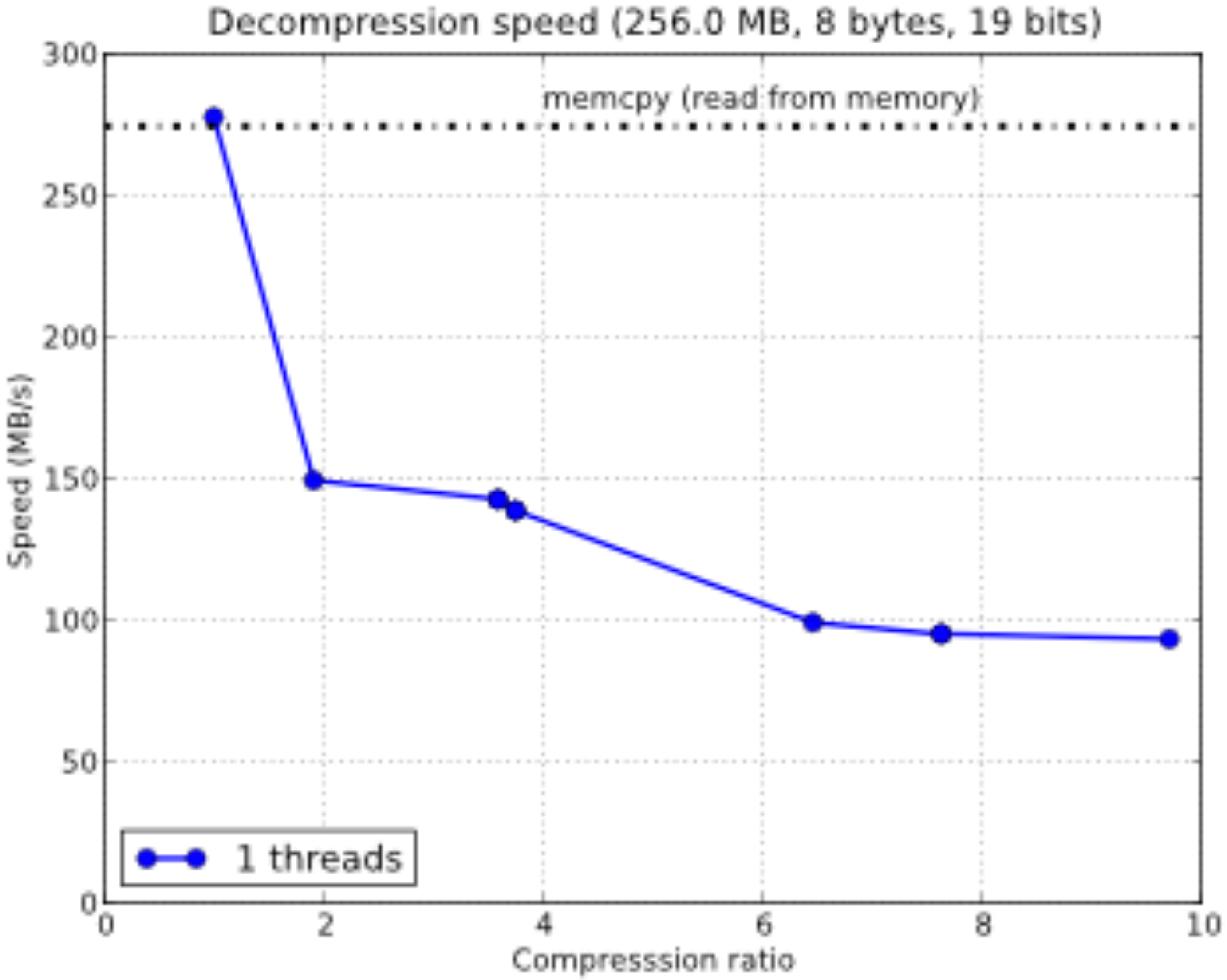


Why Blosc?

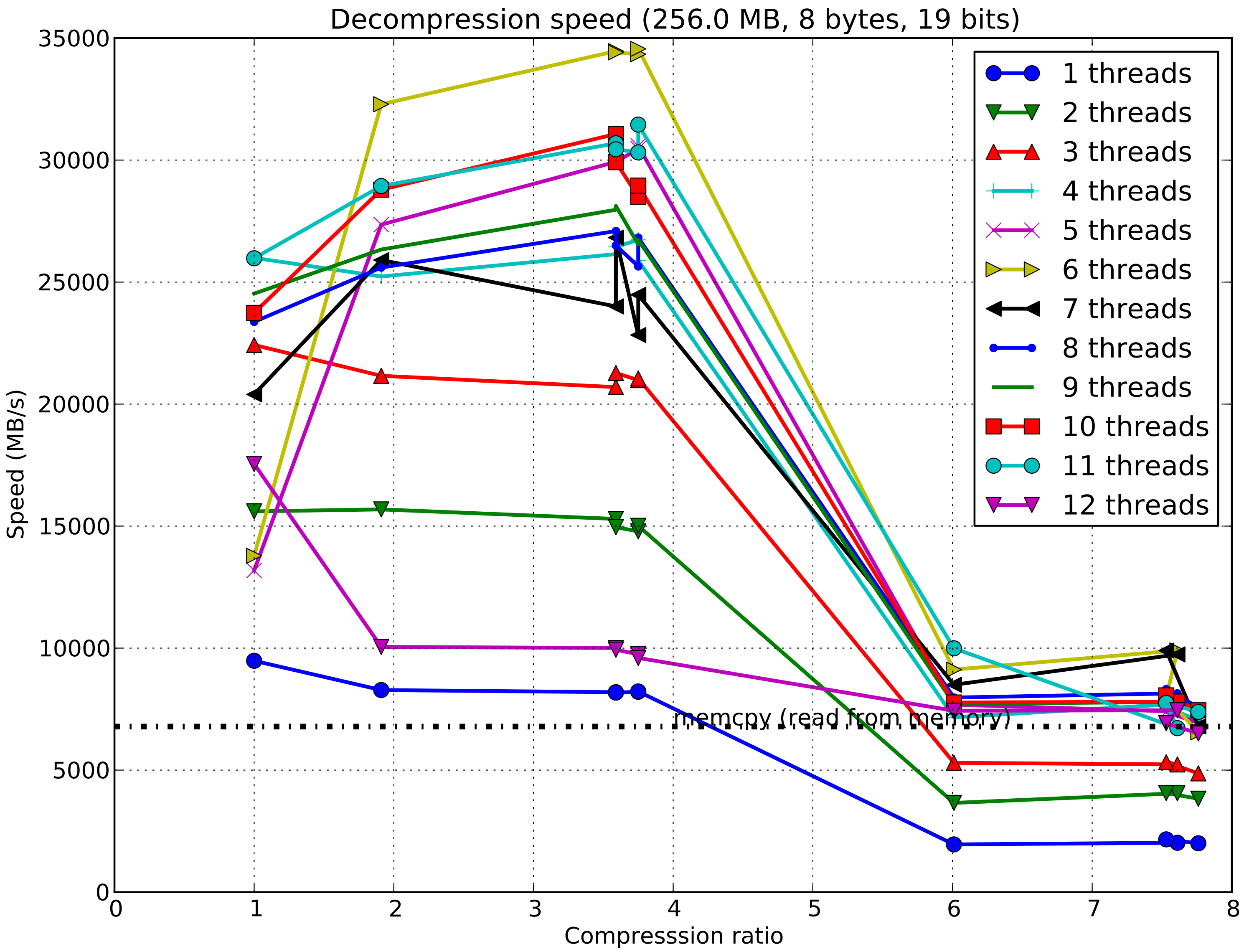


Transmission + decompression faster than direct transfer?

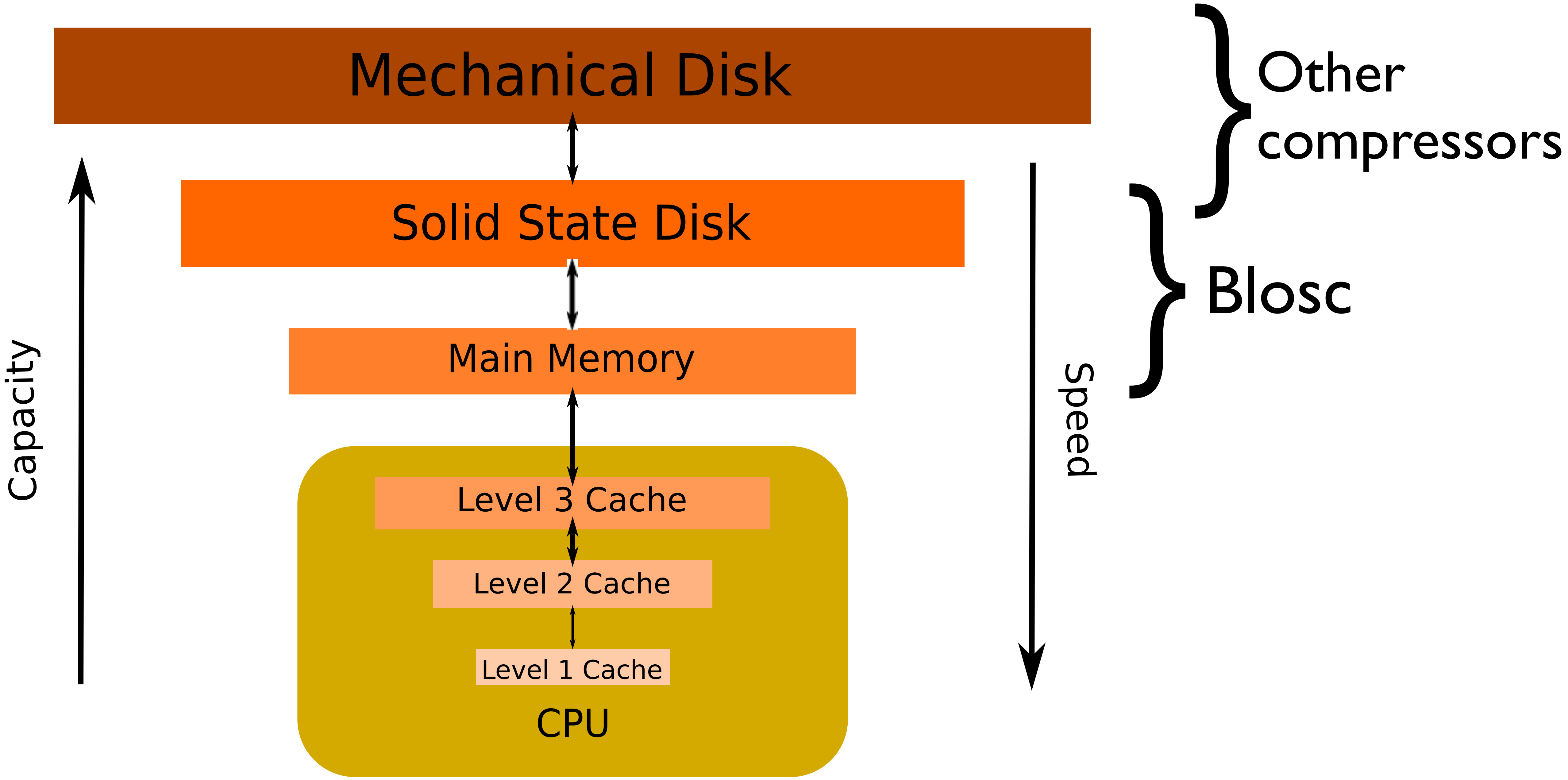
Blosc Performance: Laptop back in 2005



Blosc Performance: Desktop Computer in 2012



Accelerating I/O



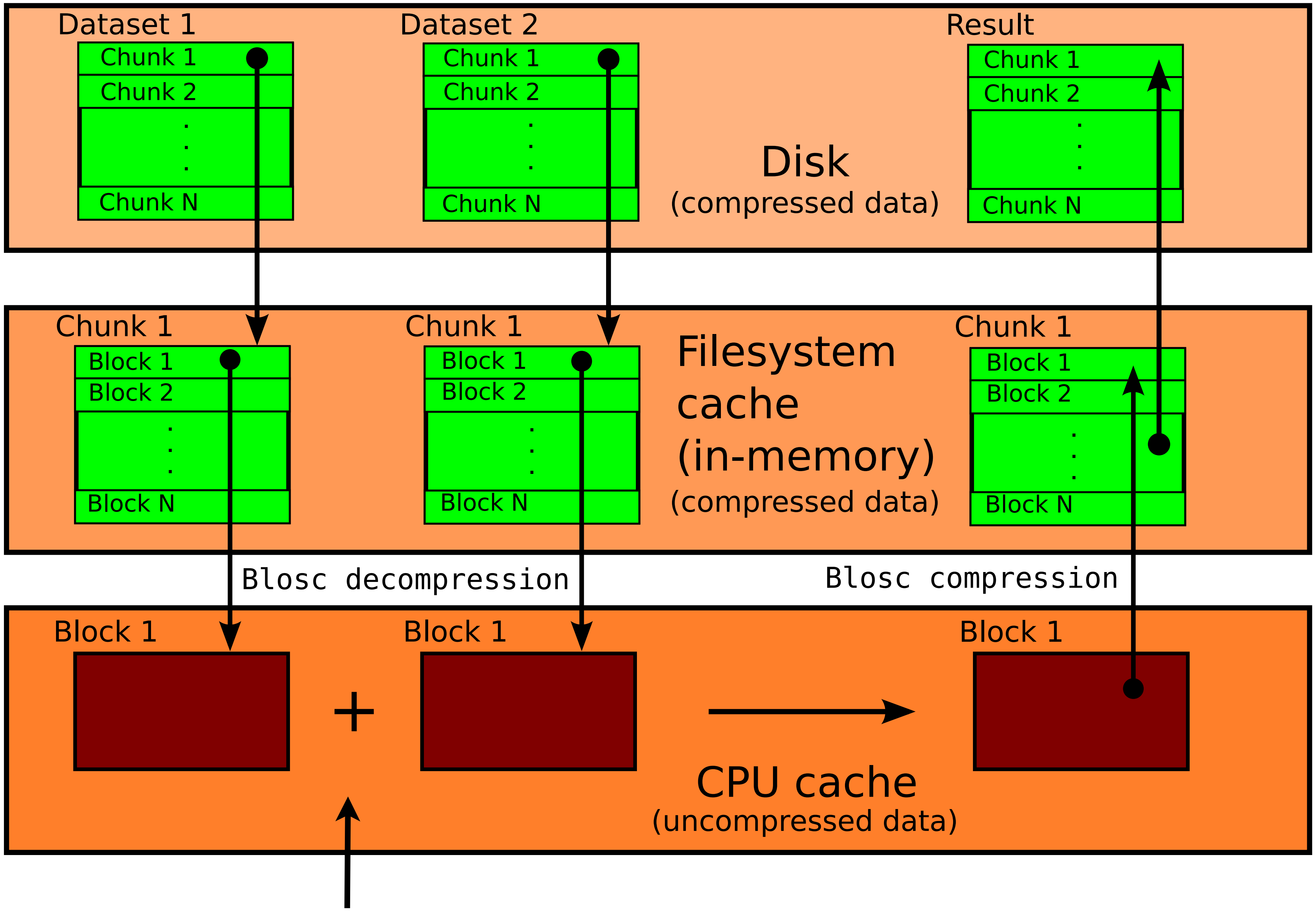
carray Objects Can Use Memory *Or* Disk

- Starting with version 0.5, carray has transparent support for data on disk too
- The format is based in ‘bloscpack’, a format for keeping data persistently (thanks to Valentin Haenel)
- To create a disk-based carray, just add the `rootdir` parameter and you are done

Out-Of-Core Computations

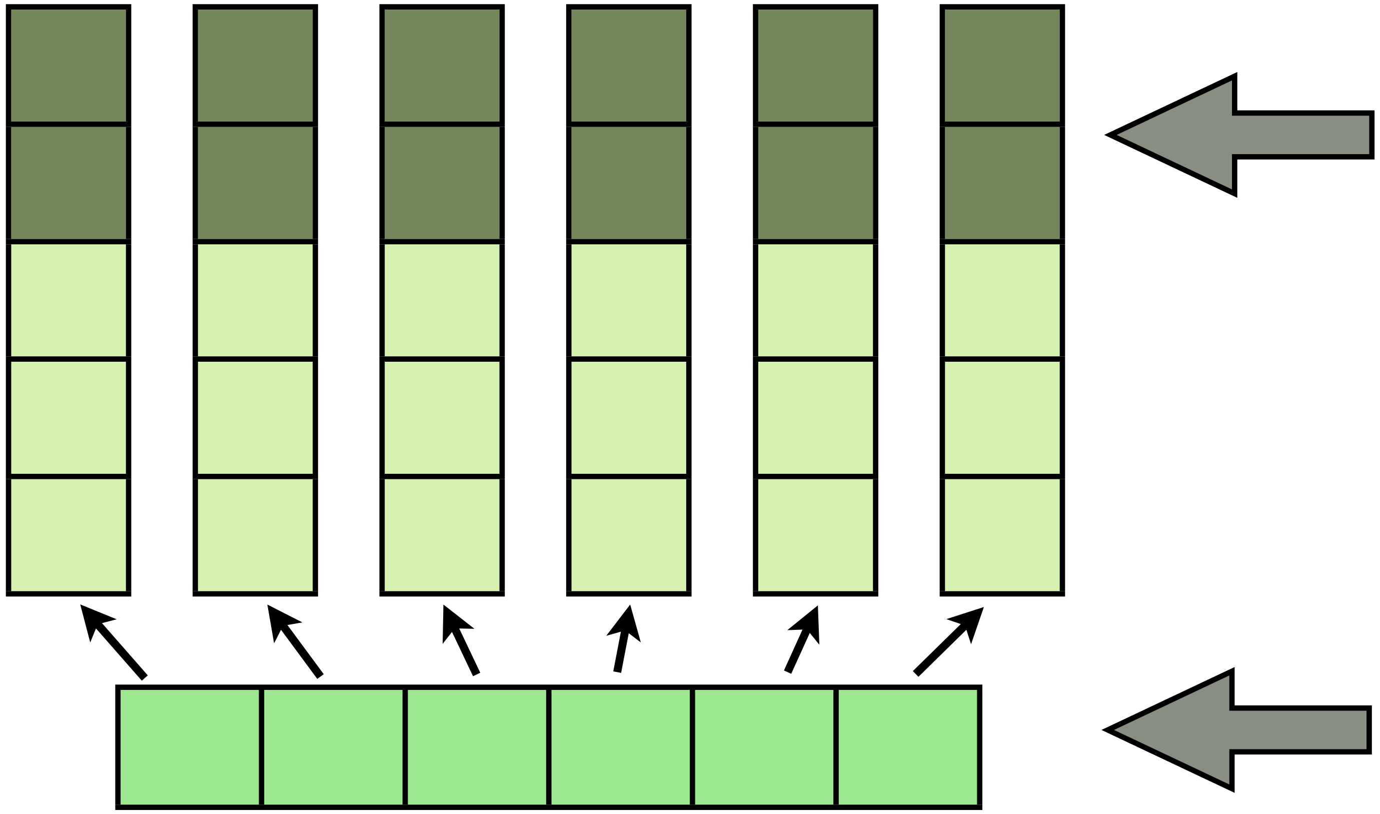
- Due to the addition of the persistency, carray can perform out-of-core computations seamlessly
- Supports different Virtual Machines:
  - Plain Python
  - numexpr (so you can use multicores)
  - Numba (in the future)

How carray Does Out-Of-Core Computations



Virtual Machine : Python, numexpr, Numba

The ctable object



Chunks

New row to append

- Columns are actual carrays
- Chunks follow **column** order
- Very efficient for querying (specially with a large number of columns)

Quick Glimpse at carray

- Creating carrays
- Making them persistent
- Operating with carrays
- Creating a ctable
- Querying ctables
- Getting results out of queries

Last Lesson for Today

- Big data is tricky to manage:

  Look for the optimal containers for you data

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

The End

Steps to Accelerate Your Code

- Make use of memory-efficient libraries (many of your bottlenecks will fall here)
- Apply the blocking technique and vectorize your code
- Parallelize (if you can) using:
  - Multi-threading
  - Explicit message passing

Summary

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

Getting More Info

- Francesc Alted — *Why modern CPUs are starving and what can be done about it*
  http://www.pytables.org/docs/CISE-12-2-ScientificPro.pdf

- David M. Cook, Francesc Alted — *How Numexpr works*
  http://code.google.com/p/numexpr/wiki/Overview

- Francesc Alted — carray manual
  http://carray.pytables.org/docs/manual

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 and carray