# Hardware-assisted speed-up techniques

## Scientific Programming with Python

Roman Gredig

University of Zurich[UZH]

# Overview

- Motivation

- The Data Access Issue
  - Why modern CPUs are starving?
  - Caches and the hierarchical memory model
  - Techniques for fighting data starvations

- High Performance Libraries

# Motivation

# Computing a Polynomial

We want to compute the polynomial:

**$y = 0.25x^3 + 0.75x^2 - 1.5x - 2$**

in the range [-1,1], with granularity of 10 million points on the x-axis

**… and we want to do that as FAST as possible …**

# use NumPy

NumPy is a powerful package that let you perform calculations with Python, but at C speed:
(see previous talks)

```
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 0.86 sec on our machine (Intel Core i5-3380M CPU @ 2.90GHz).

Hint: use `%timeit` in ipython for easy benchmarking

**How to make it faster?**

# "Quick & Dirty" Approach: Parallelize

- The problem of 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, for example, using the multiprocessing module. See **`poly-mp.py`** script.

- Using 2 cores, the 0.86 sec is slowed down to 0.88 sec! WTF?

- Why do I even buy a multi-core computer?

# Another (Much Easier) Approach: Factorize

- The NumPy expression:
  (I) `y = .25*x**3 + .75*x**2 - 1.5*x - 2`

  can be written as

  (II) `y = ((.25*x + .75)*x - 1.5)*x - 2`

- With this, the time goes from 0.86 sec to 0.107 sec, which is much faster (8x) than using two processors with the **`multiprocessing`** approach (0.88 sec).

**Give optimization a chance before parallelizing!**

# Numexpr Can Compute Expressions Way Faster

- Numexpr is a just-in-time (JIT) compiler, based on NumPy, that optimizes the evaluation of complex expressions. Its use is easy:

```
import numpy as np
import numexpr as ne
N = 10*1000*1000
x = np.linspace(-1, 1, N)
ne.set_num_threads(1) # use only one thread/cpu
y = ne.evaluate('.25*x**3 + .75*x**2 - 1.5*x - 2')
```

- That takes around 0.059 sec to complete, which is 15x faster than the original NumPy expression (0.86 sec).

# Fine-tune Expressions with Numexpr

- Numexpr is also sensible to computer-friendly expressions like:

  (II) `y = ((.25*x + .75)*x - 1.5)*x - 2`

- Numexpr takes 0.046 sec for the above (0.059 sec were needed for the original expression, that's a 28% faster)

# Using Multiple Threads with Numexpr

- Numexpr supports multi processing:

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

- That takes around 0.029 sec to complete, which is a 60% faster than using a single processor (0.046 sec).

# Summary and Open Questions

| | 1 core | 2 cores | Parallel Speedup |
|---|---|---|---|
| **NumPy (I)** | 0.876 | 0.877 | 0.98x |
| **NumPy (II)** | 0.107 | 0.484 | 0.22x |
| **Numexpr (I)** | 0.059 | 0.034 | 1.74x |
| **Numexpr (II)** | 0.046 | 0.029 | 1.59x |

- If all the approaches perform the same computations, all in C space, why the wild differences in performance?

- Why the different approaches do not scale similarly in parallel mode?

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

**(II)**
```
y = ((.25*x + .75)*x - 1.5)*x - 2
```

# A First Answer:
# Power Expansion and Performance

Numexpr expands the 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 - 2
```

so, no need to use the expensive **`pow()`**

# One (Important) Remaining Question

Why can numexpr execute this expression:

```
((.25*x + .75)*x - 1.5)*x - 2
```

more than 2x faster than NumPy?

# One (Important) Remaining Question

Why can numexpr execute this expression:

```
((.25*x + .75)*x - 1.5)*x - 2
```

more than 2x faster than NumPy?

**By making a more efficient use of the memory resource**

# The Data Access Issue

# 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

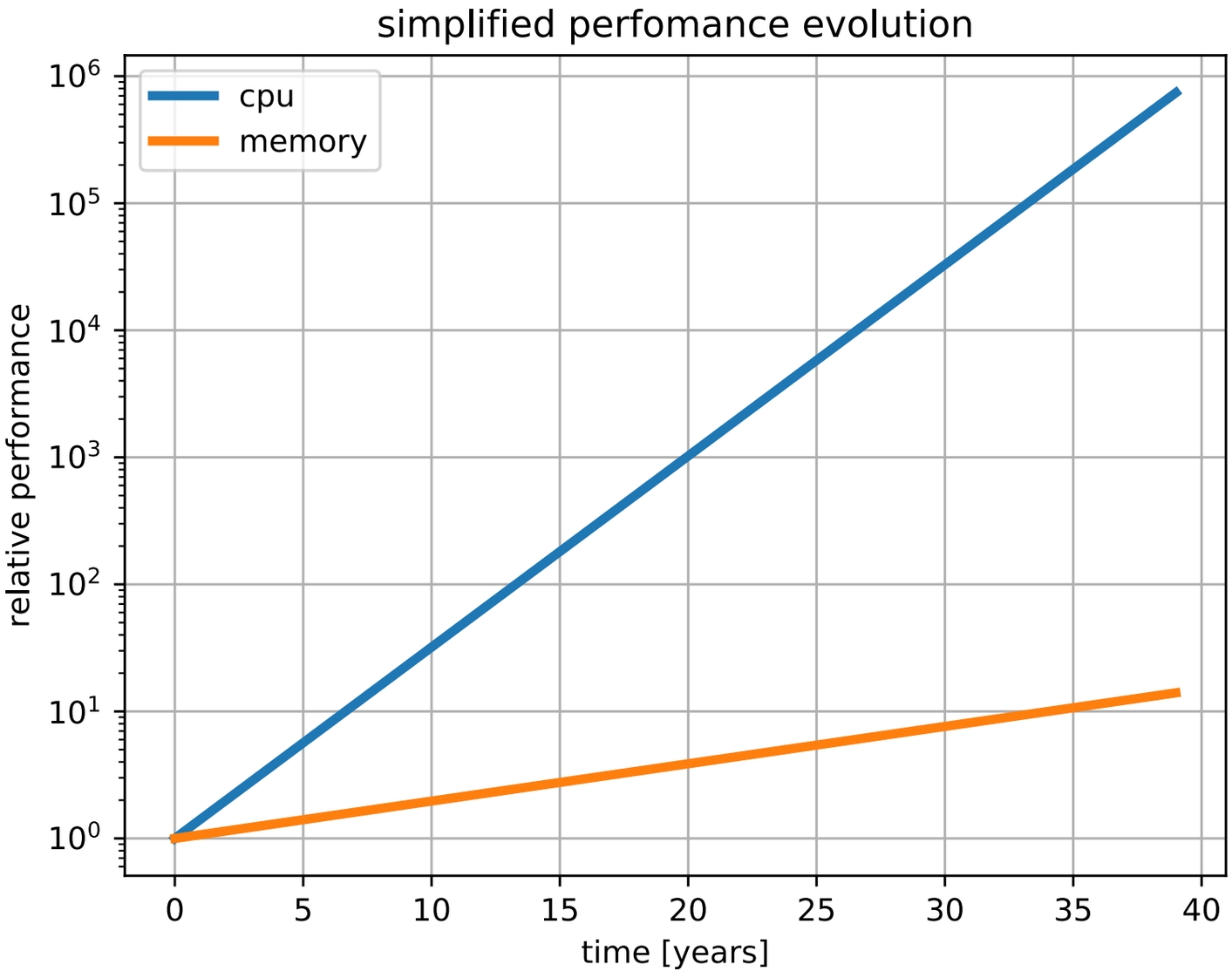
# Quote Back in 1996

“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.”**

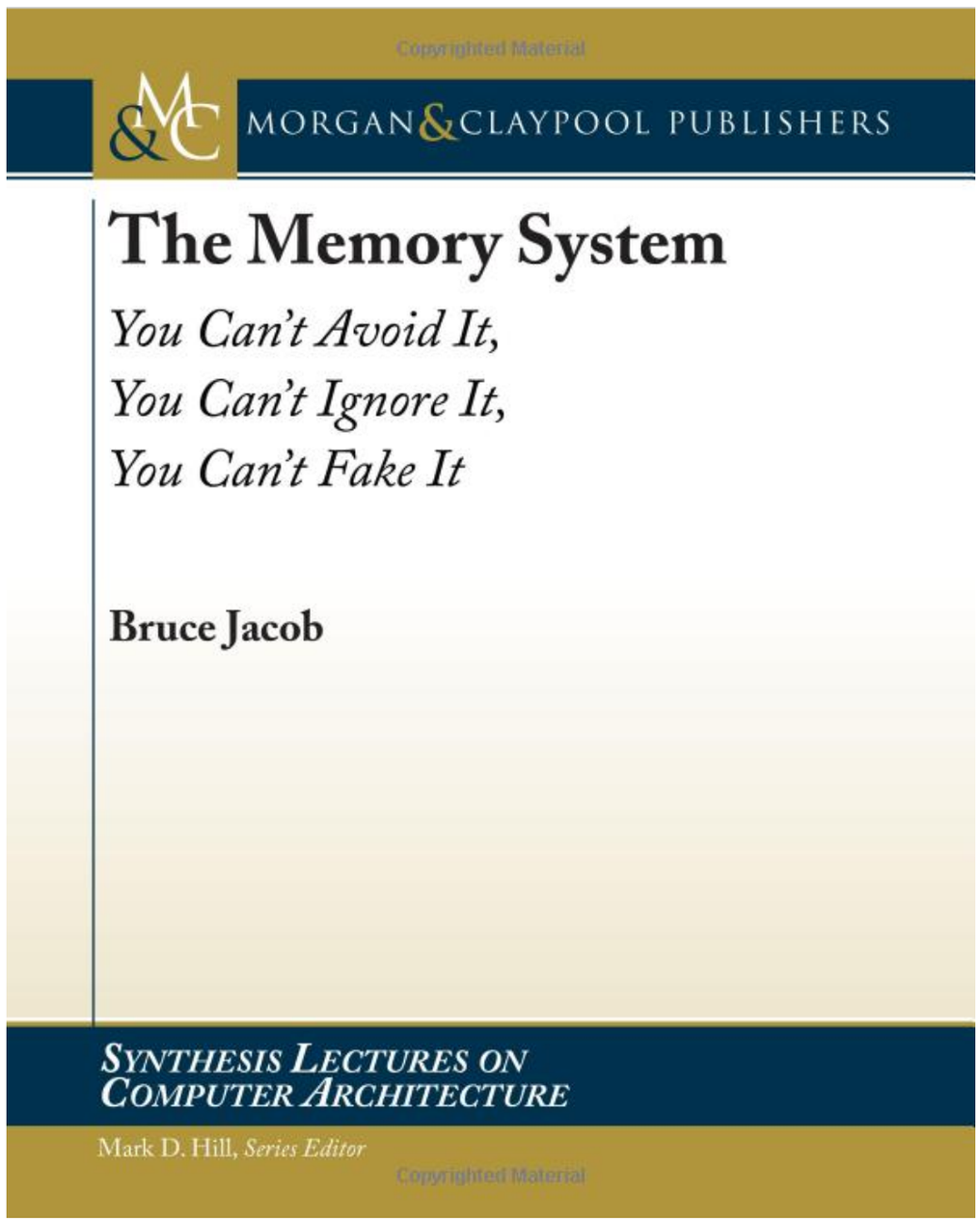
– Richard Sites, after his article “It’s The Memory, Stupid!”, Microprocessor Report, 10(10),1996

# CPU vs. Memory Cycle Trend




CPU speed increases much faster than memory speed → **performance gap**

# 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 CPU Starvation Problem

- **Memory latency** is much slower (between 250x and 500x) than processors and is an essential bottleneck**.**
- **Memory throughput** is improving at a better rate than memory latency, but it is also much slower than processors (between 30x and 100x)**.**

The result is that CPUs in our current computers are suffering from a serious starvation data problem:

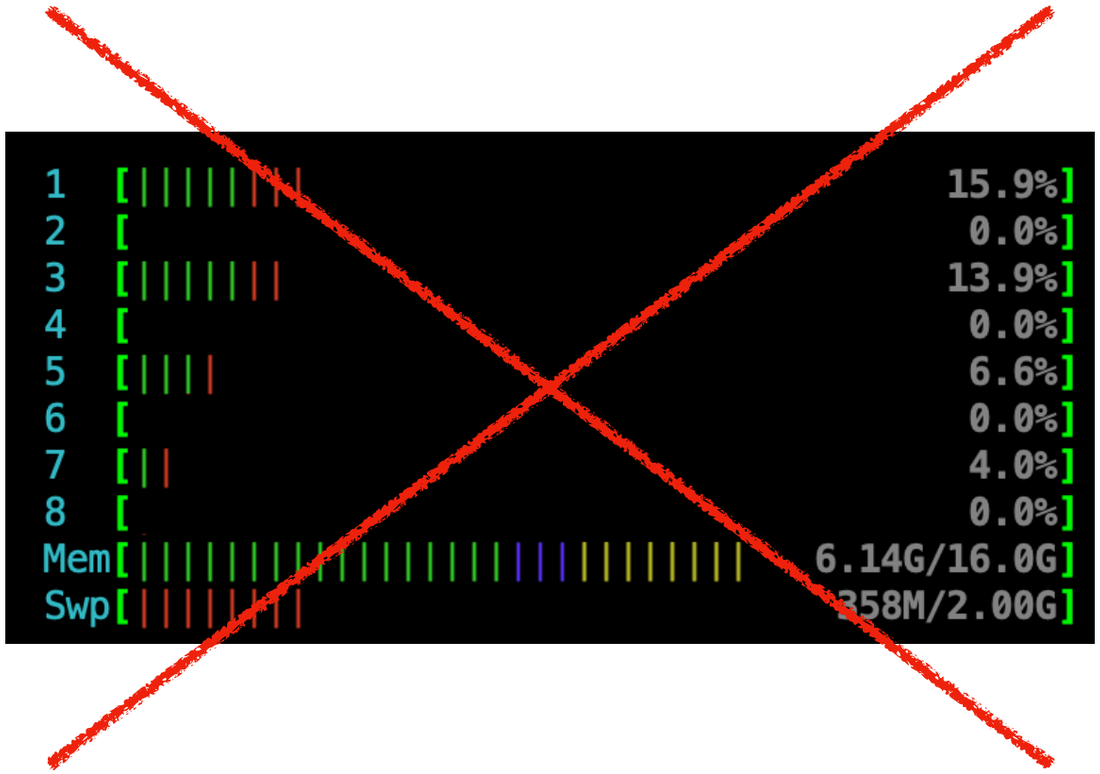
**They could consume (much!) more data than the system can possibly deliver.**

# What Is the Industry Doing to Alleviate CPU Starvation?

- They are improving memory throughput: cheaper to implement (more data is transmitted on each clock cycle).
- They are adding big **caches** in the CPU die (i.e. the “chip”).
    - Different types of memory:
        - regular memory: dynamic RAM (DRAM)
            - cheap
            - dense
            - needs periodic refresh
- cache memory: static RAM (SRAM)
    - more complex
    - no refresh needed
    - increased power consumption at faster access

# What Is the Industry Doing to Alleviate CPU Starvation?

- They are improving memory throughput: cheaper to implement (more data is transmitted on each clock cycle).
- They are adding big **caches** in the CPU die (i.e. the “chip”).
  - Different types of memory:
    - regular memory: dynamic RAM (DRAM)
      - cheap
      - dense
      - needs periodic refresh
    - cache memory: static RAM (SRAM)
      - more complex
      - no refresh needed
      - increased power consumption



# Why Is a Cache Useful?

- Caches are closer to the processor (normally in the same die), so both the latency and throughput are improved.

- However: the faster they run the smaller they must be.

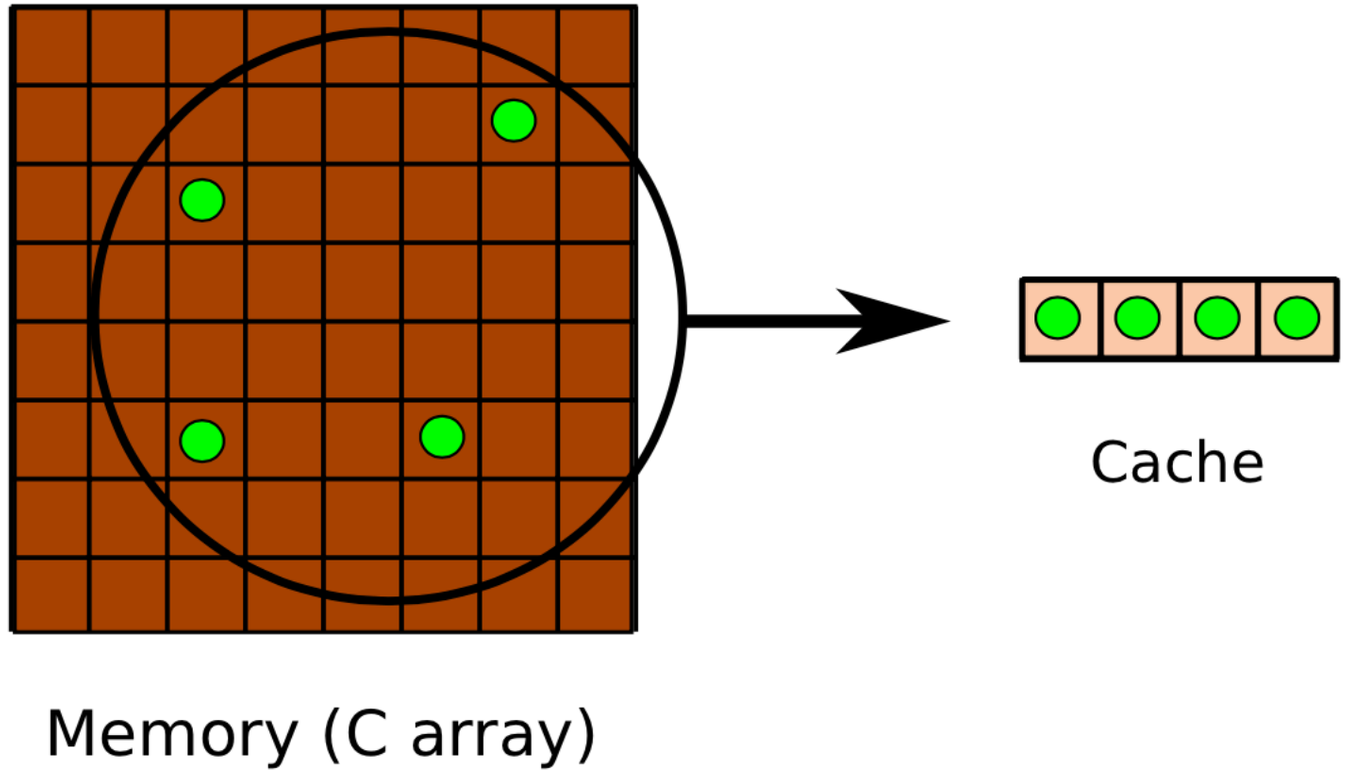
- They are effective mainly in a couple of scenarios:

  - **Time locality**: when the dataset is reused.

  - **Spatial locality**: when the dataset is accessed sequentially.

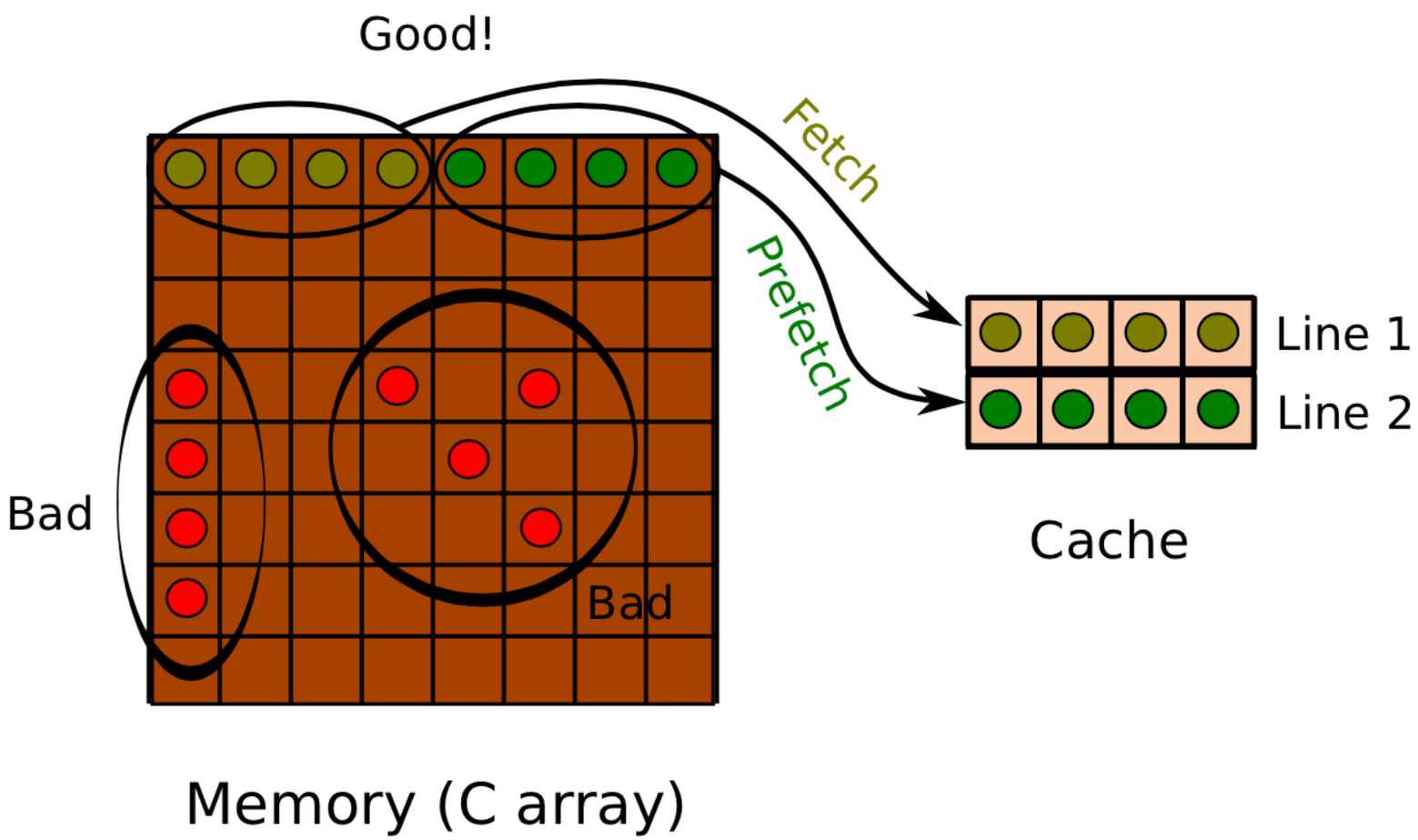
# Time Locality

Parts of the dataset are reused



# Space Locality

Dataset is accessed sequentially

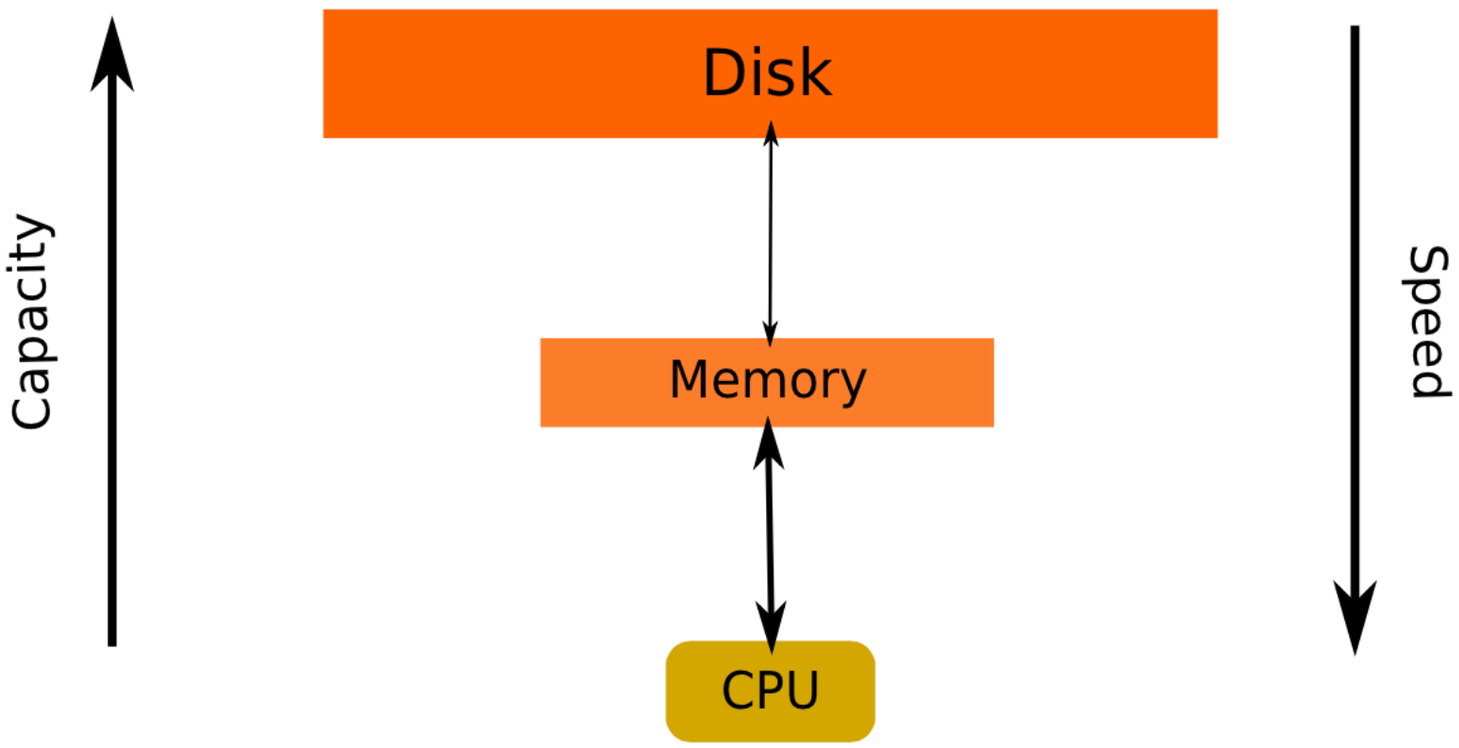


# The Hierarchical Memory Model

- Introduced by industry to cope with CPU data starvation problems.
- It consists in having several layers of memory with different capabilities:
    - Lower levels (i.e. closer to the CPU) have higher speed, but reduced capacity. Best suited for performing computations.
    - Higher levels have reduced speed, but higher capacity. Best suited for storage purposes.

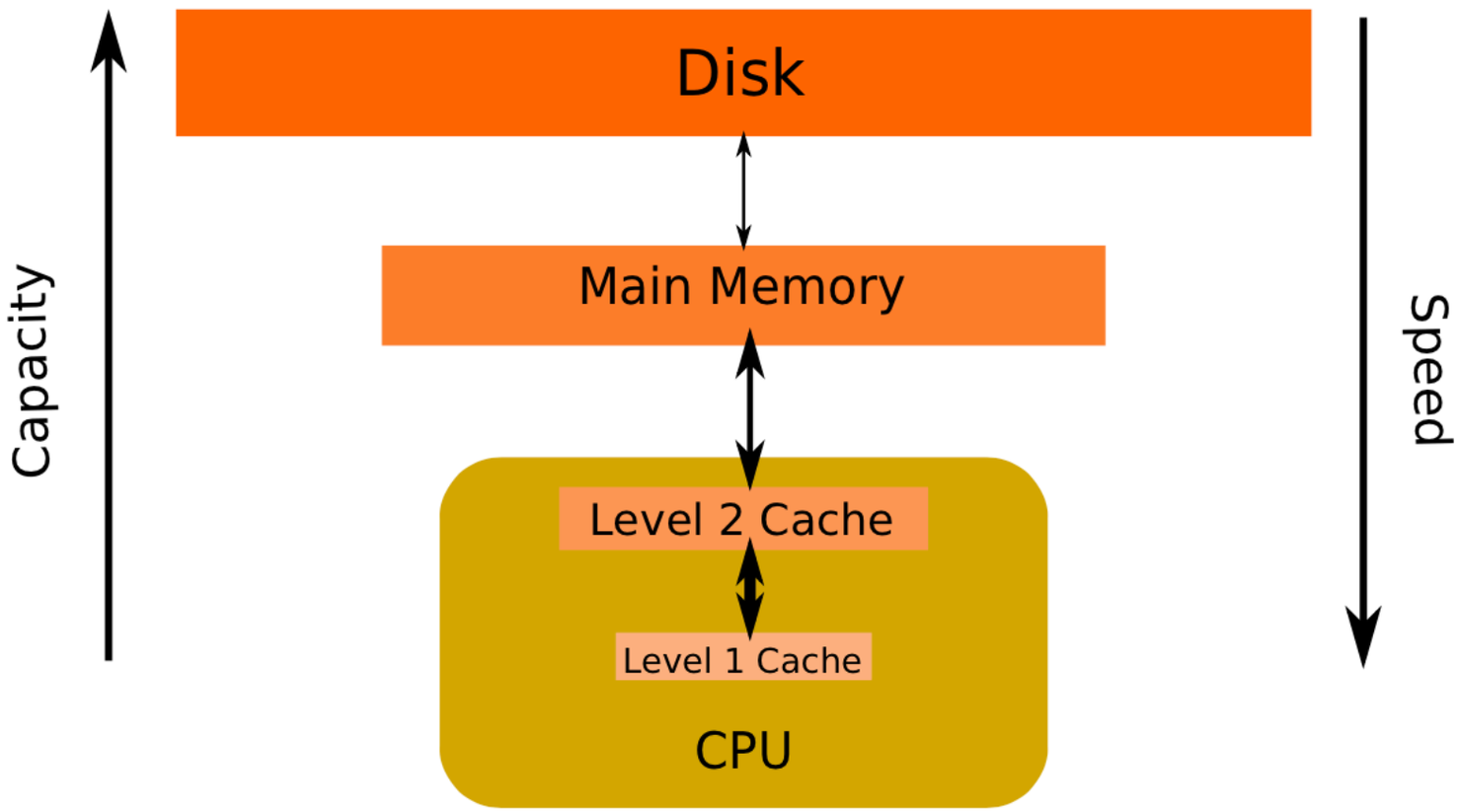
# The Primordial Hierarchical Memory Model

Two level hierarchy:



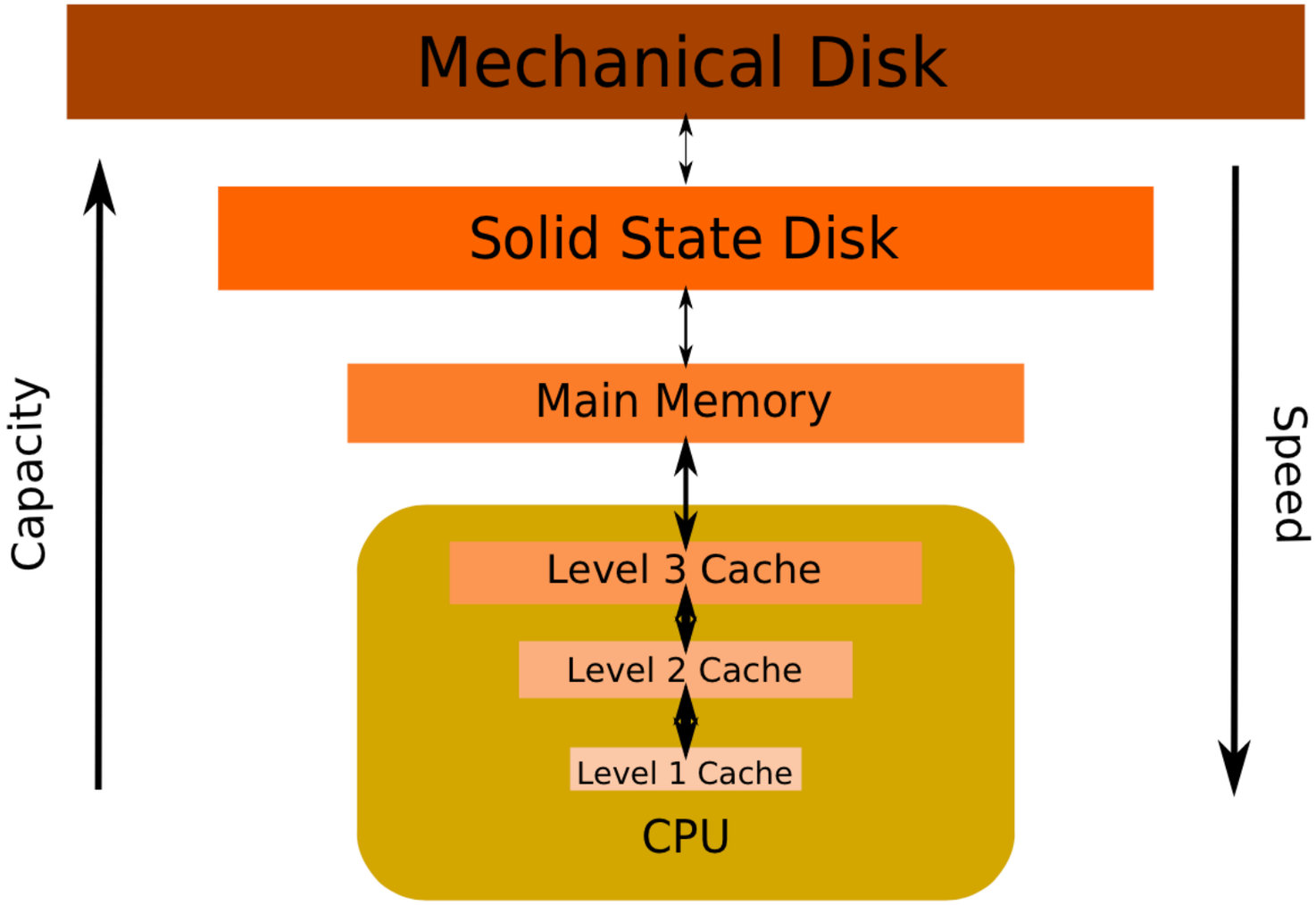
# The 2000's Hierarchical Memory Model

Four level hierarchy:



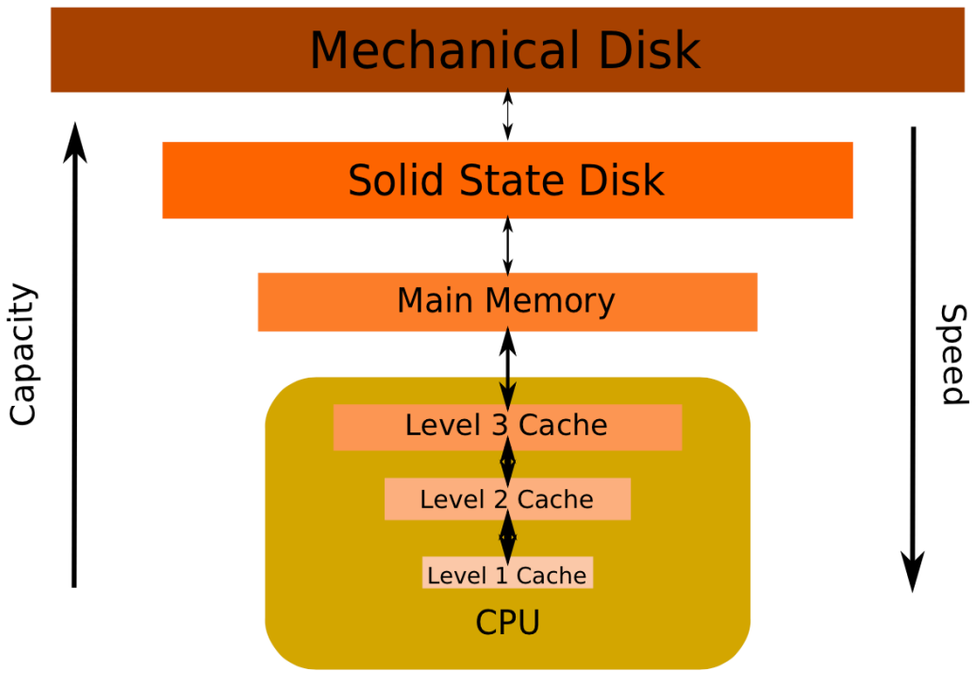
# The Current Hierarchical Memory Model

Six level (or more) hierarchy:

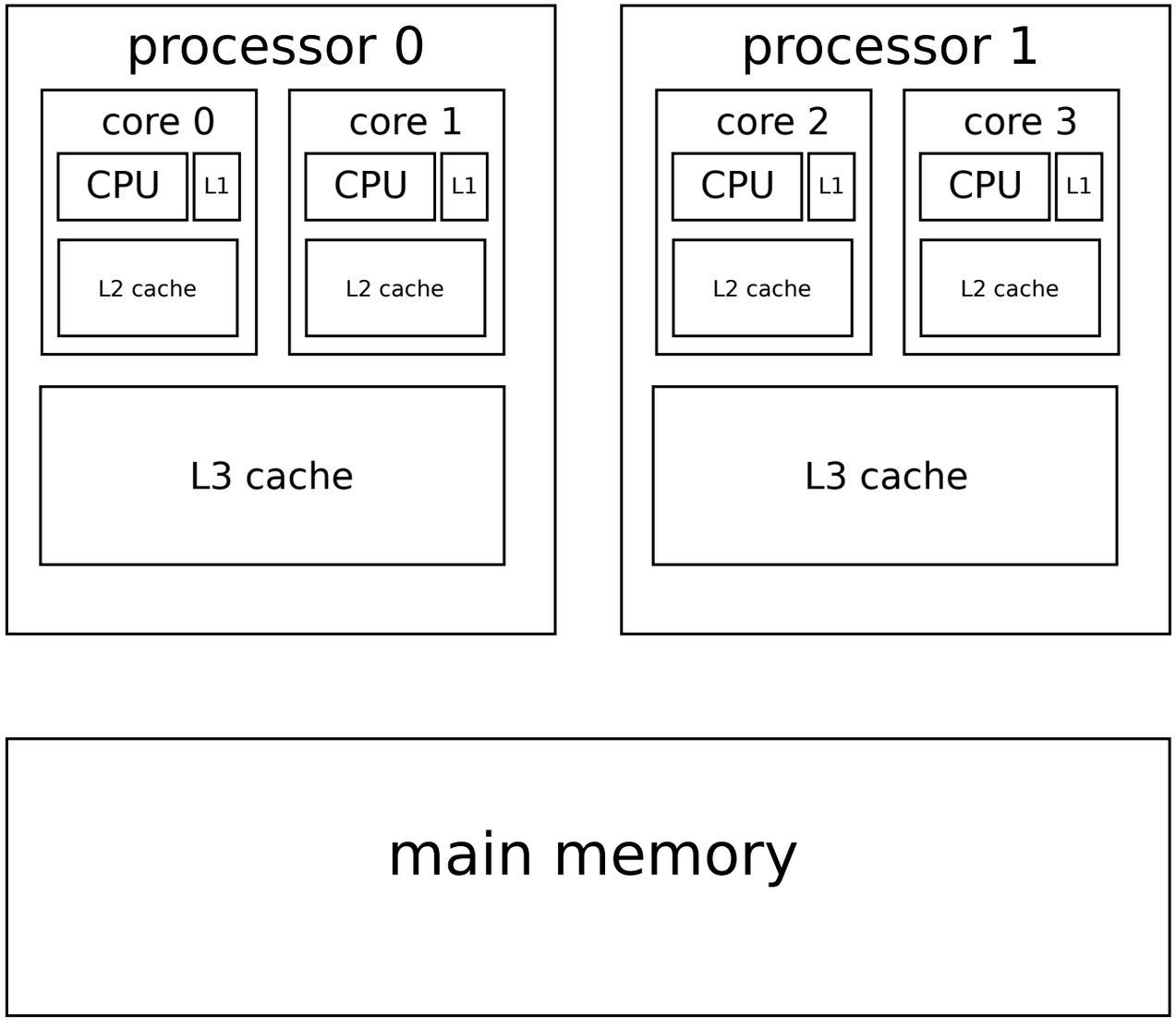


# The Current Hierarchical Memory Model

Six level (or more) hierarchy:




one example (layouts differ a lot):



# Once Upon A Time …

- In the 1970s and 1980s many computational scientists had to learn assembly language in order to squeeze all the performance out of their processors.

- "written in assembler" used to be an advertisement

- In the good old days, the processor was the key bottleneck.

# Nowadays …

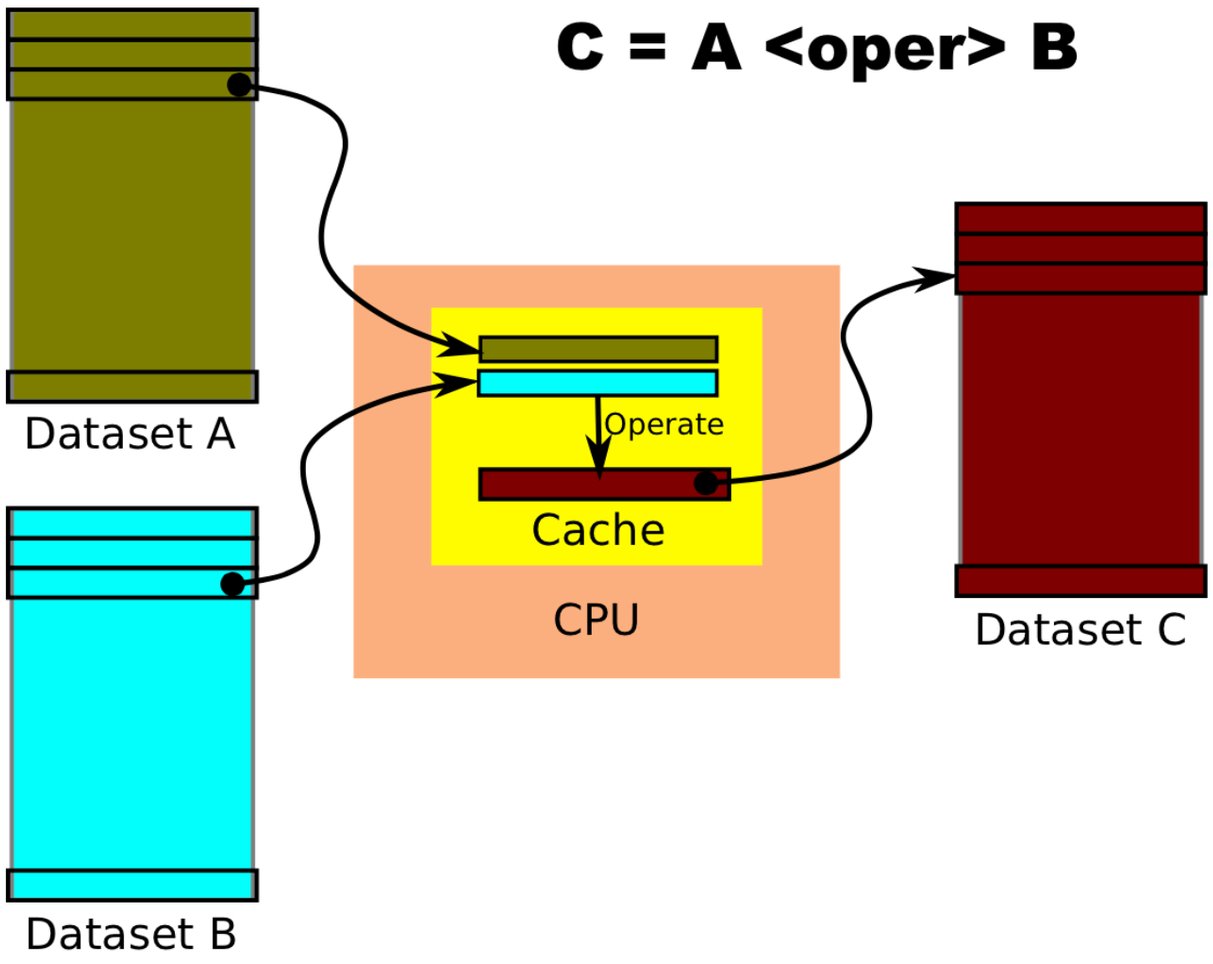
- Every computer scientist must acquire a good knowledge of the hierarchical memory model (and its implications) if they want their applications to run at a decent speed (i.e. they do not want their CPUs to starve too much).

- Memory organization has become now the key factor for optimizing.

- You don't need to know how to put data in the cache, but help the OS to do it efficiently.

# The Blocking Technique

When you have to access memory, get a contiguous block that fits in the CPU cache, operate upon it or reuse it as much as possible, then write the block back to memory:



# Understand NumPy Memory Layout

Being **a** a squared array (4000x4000) of doubles, we have:

Summing up column-wise:

```
a[:,1].sum()      # takes 9.3 ms
```

Summing up row-wise: more than 100x faster (!)

```
a[1,:].sum()      # takes 72 µs
```

**NumPy arrays are ordered row-wise (C convention) by default**

When would **a[1,:].sum()** be slower than **a[:,1].sum()**?

# Vectorize Your Code

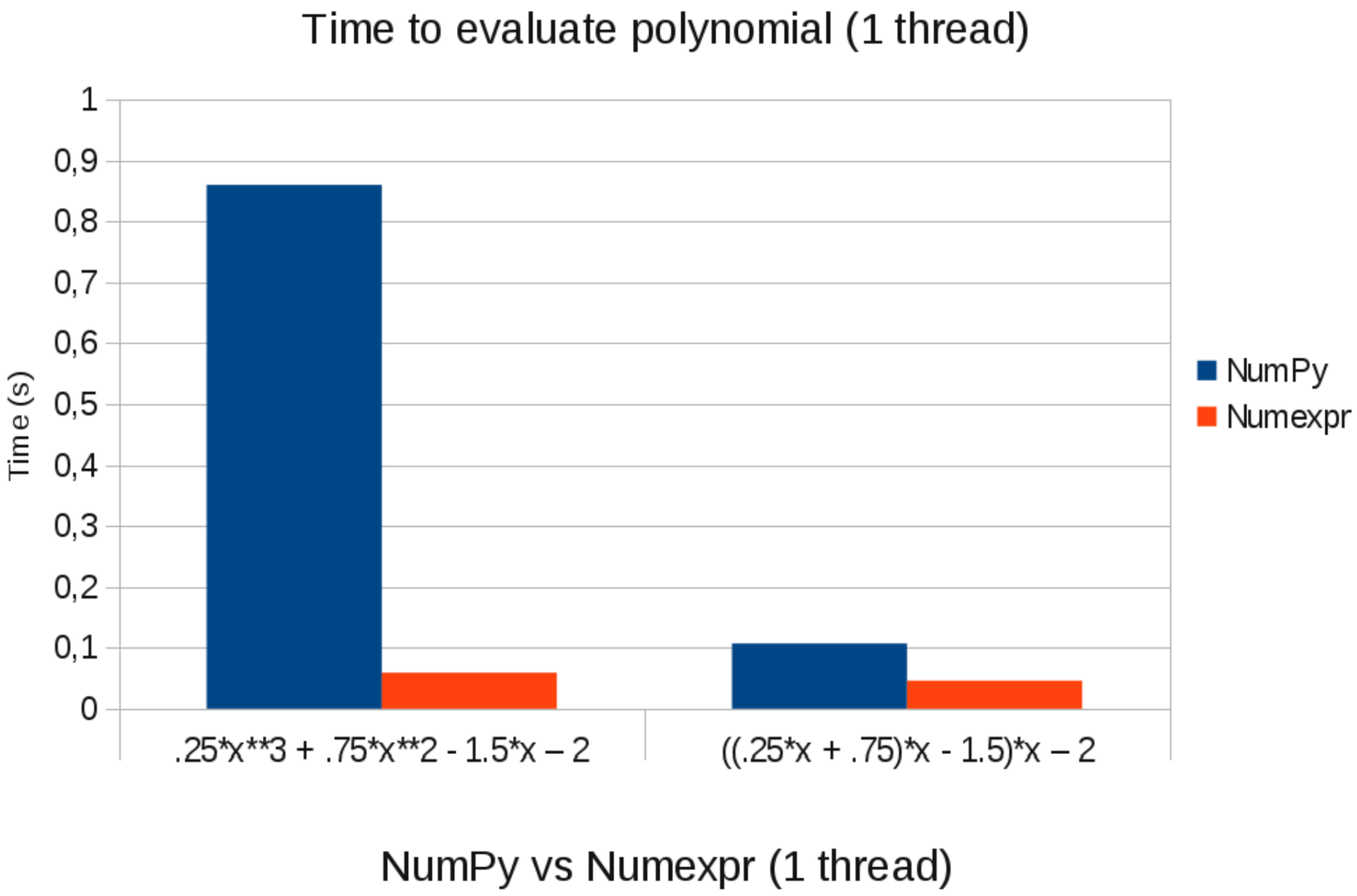
Naive matrix-matrix multiplication: 1264 s (1000x1000 doubles)

```
def dot_naive(a,b):
    c = np.zeros((nrows, ncols), dtype='f8')
    for row in range(nrows):
        for col in range(ncols):
            for i in range(nrows):
                c[row,col] += a[row,i] * b[i,col]
    return c
```

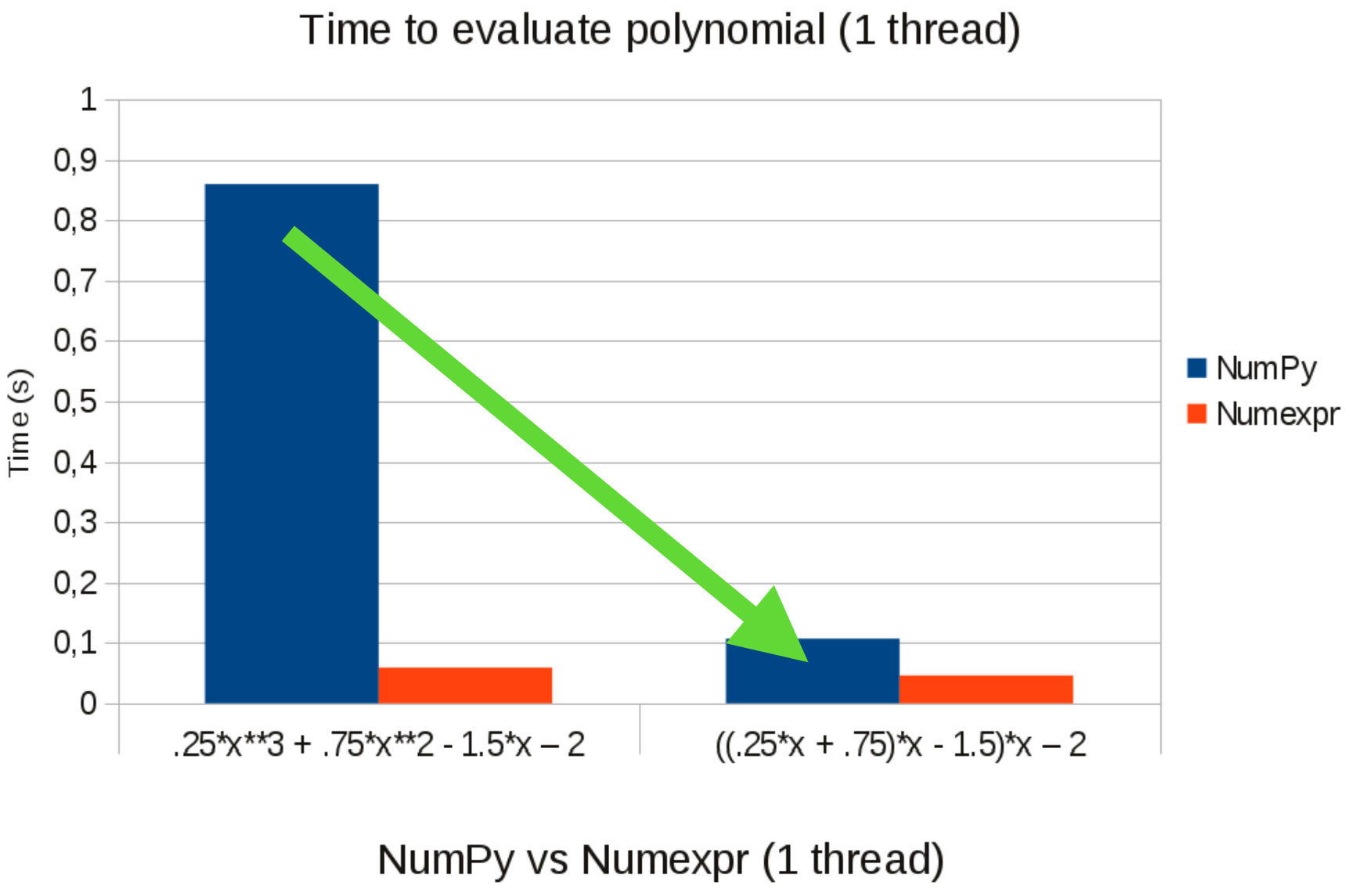
Vectorized matrix-matrix multiplication: 20 s (64x faster)

```
def dot(a,b):
    c = np.empty((nrows, ncols), dtype='f8')
    for row in range(nrows):
        for col in range(ncols):
            c[row, col] = np.sum(a[row] * b[:,col])
    return c
```

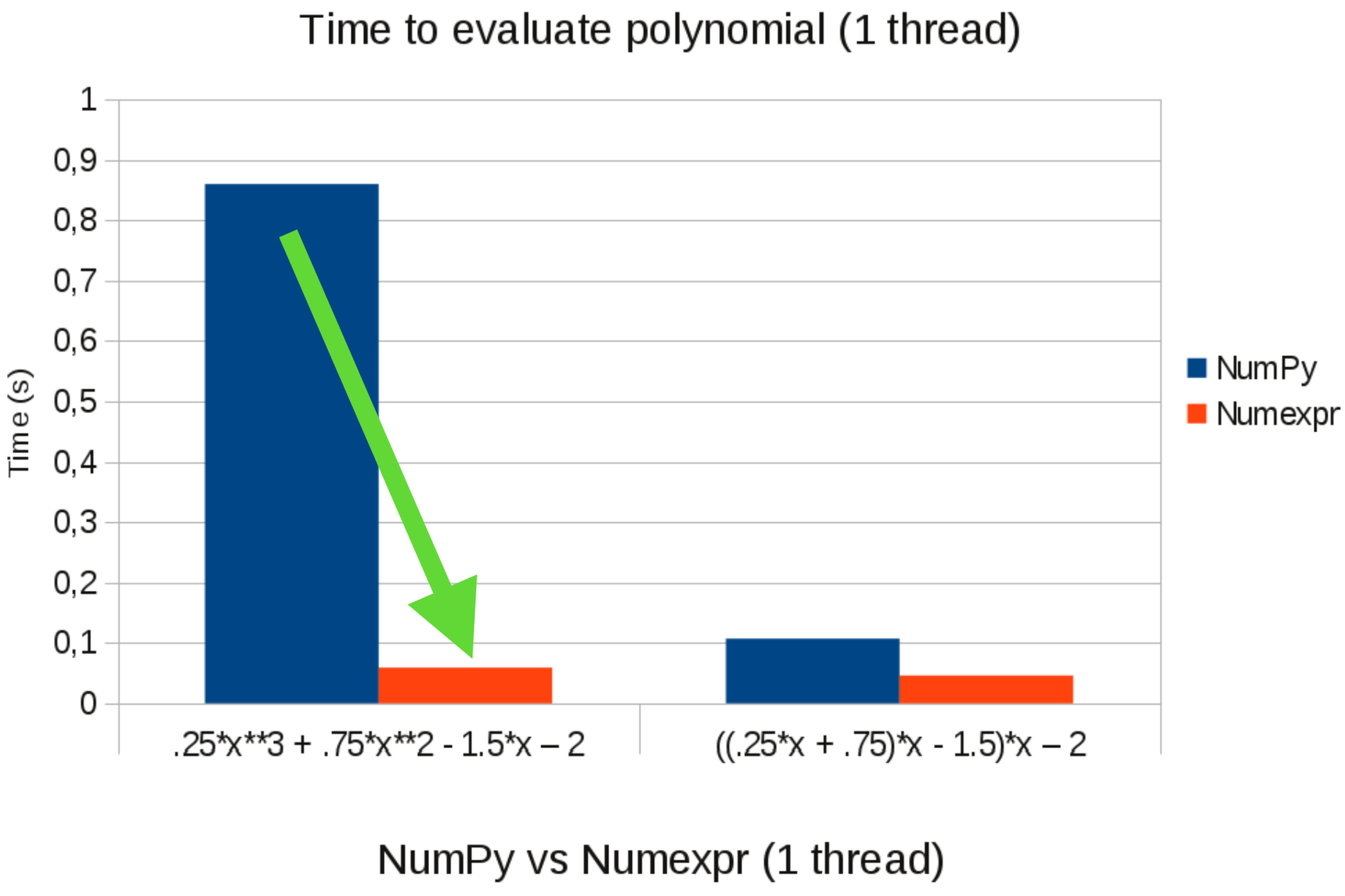
# Interlude: Resolving More Open Questions



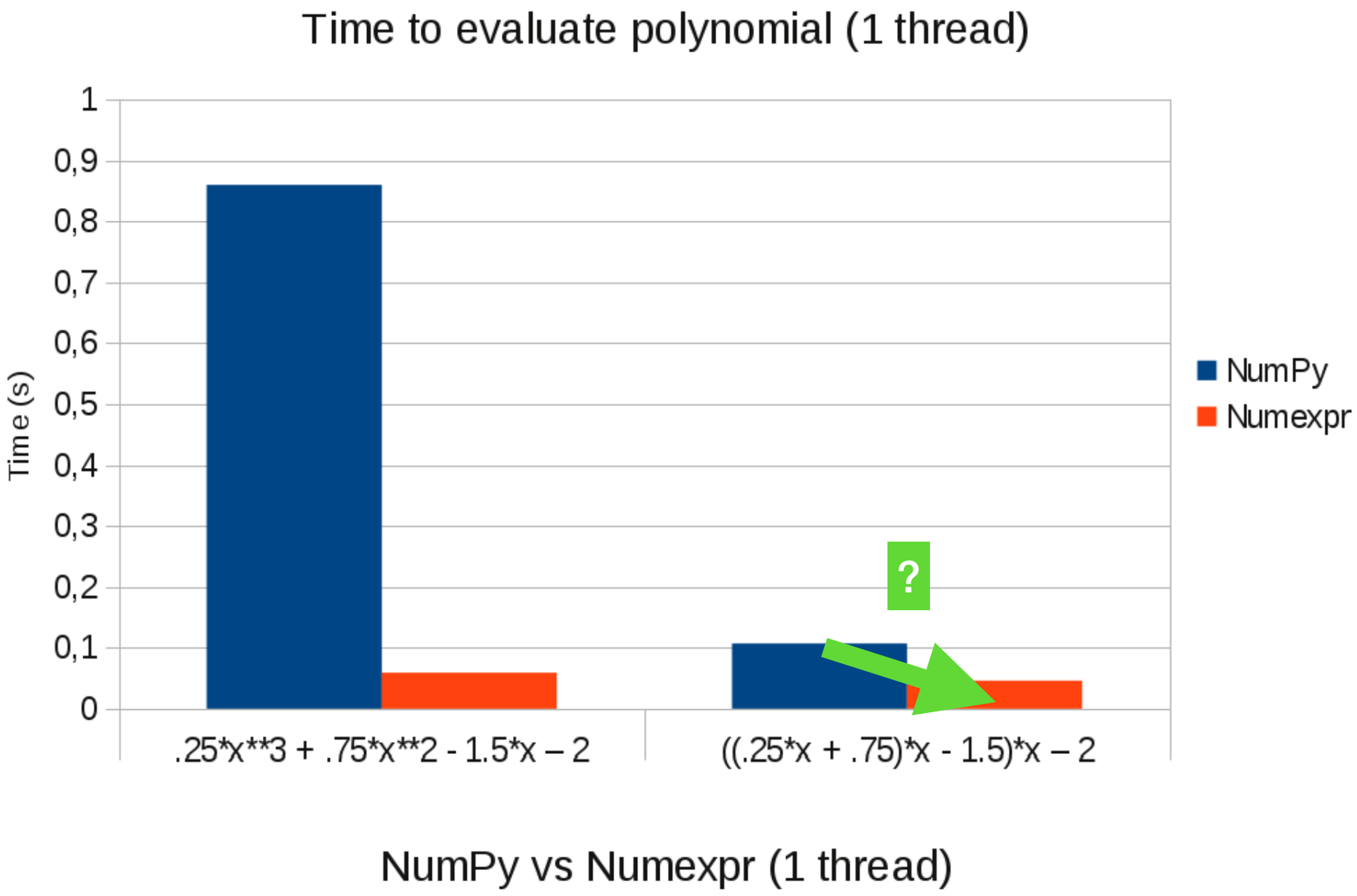
# Interlude: Resolving More Open Questions



# Interlude: Resolving More Open Questions

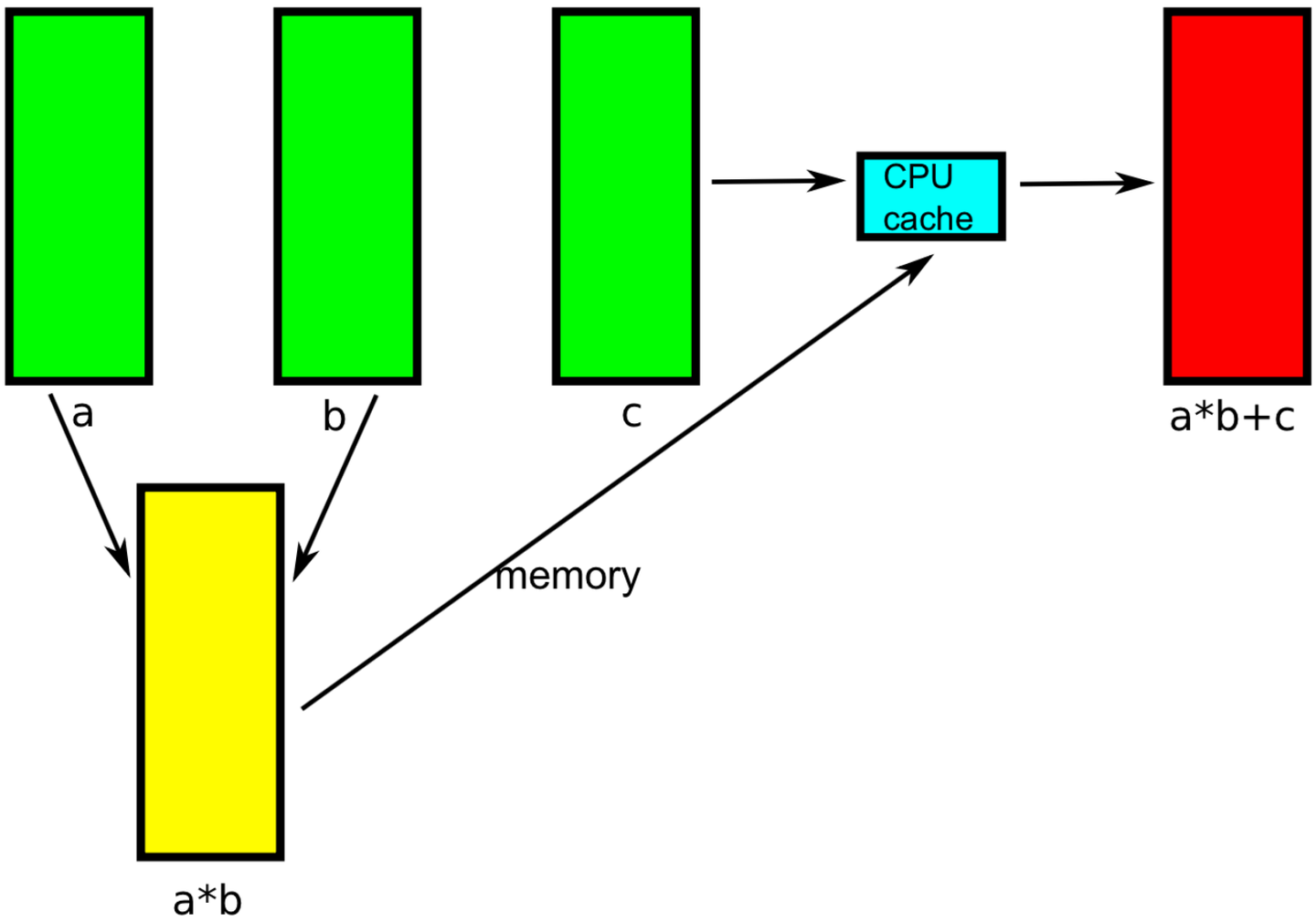


# Interlude: Resolving More Open Questions



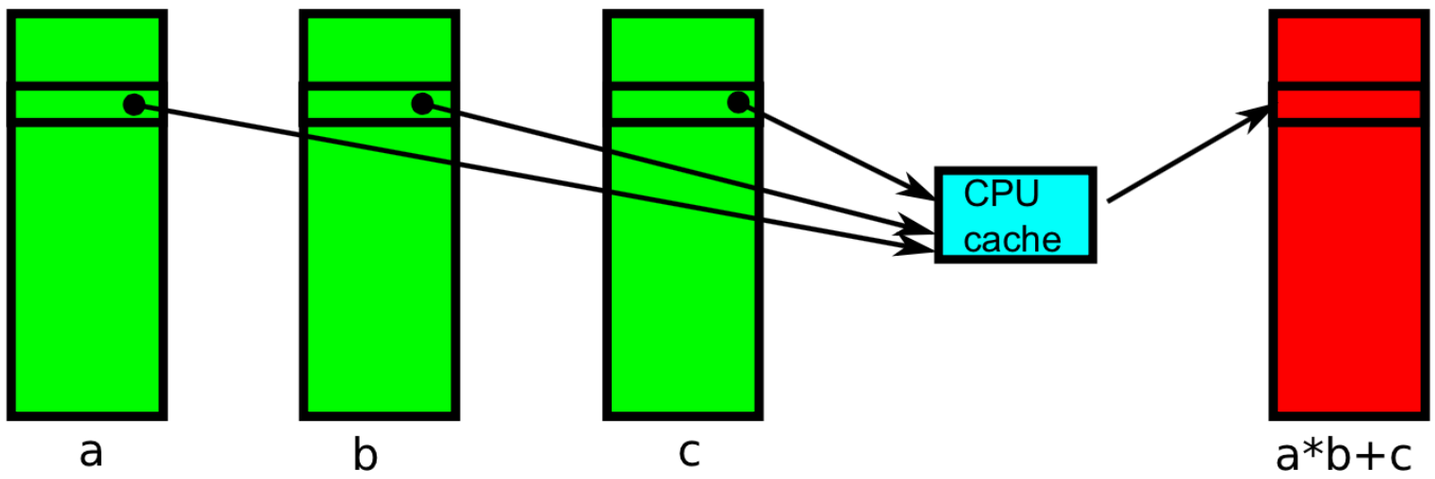
# NumPy And Temporaries

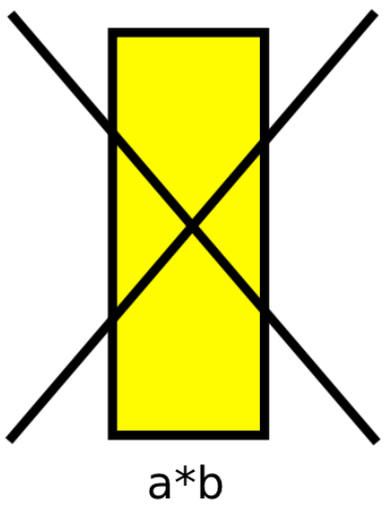
Computing "a*b+c" with NumPy. Temporaries goes to memory.



# Numexpr Avoids (Big) Temporaries

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




memory

# Mysteries (almost) Solved Now

## But why does is scale so differently ?

| | 1 core | 2 cores | Parallel Speedup |
|---|---|---|---|
| **NumPy (I)** | 0.876 | 0.877 | 0.98x |
| **NumPy (II)** | 0.107 | 0.484 | 0.22x |
| **Numexpr (I)** | 0.059 | 0.034 | 1.74x |
| **Numexpr (II)** | 0.046 | 0.029 | 1.59x |

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

**(II)**
```
y = ((.25*x + .75)*x - 1.5)*x - 2
```

# Numba: Overcoming numexpr Limitations

- Numba is a JIT compiler that can translate a subset of the Python language into machine code

- For a single thread, it can achieve similar or better performance than numexpr, but with more flexibility

- The costs of compilation can be somewhat high though

- Free software

# Numba Example: Computing the Polynomial

```
from numba import jit
import numpy as np
N = 10*1000*1000
x = np.linspace(-1, 1, N)
y = np.empty(N, dtype=np.float64)

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

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

# Times for Computing the Polynomial

| | 1 core | 2 cores | Parallel Speedup |
|---|---|---|---|
| **NumPy (I)** | 0.876 | 0.877 | 0.98x |
| **NumPy (II)** | 0.107 | 0.484 | 0.22x |
| **Numexpr (I)** | 0.059 | 0.034 | 1.74x |
| **Numexpr (II)** | 0.046 | 0.029 | 1.59x |
| **Numba (I)** | 0.731 | | |
| **Numba (II)** | 0.037 | | |

Compilation time for Numba: 0.321 sec

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

**(II)**
```
y = ((.25*x + .75)*x - 1.5)*x - 2
```

# Steps To Accelerate Your Code

In order of importance:

- Make use of memory-efficient libraries (many of the current bottlenecks fall into this category).

- Apply the blocking technique and vectorize your code.

- Parallelize using:

    - Multi-threading (using Cython, numexpr, …).

    - Multi-processing (via the multiprocessing module in Python)

    - Explicit message passing (IPython, MPI via mpi4py).

**Parallelization is usually a pretty complex thing to program, so let's use existing libraries first!**

# Summary

- These days, you should understand the hierarchical memory model if you want to get decent performance.
- Existing memory-efficient libraries help you to perform your computations optimally.
- Do not blindly try to parallelize immediately. Do this as a last resort!

# More Info

- Ulrich Drepper:
  What Every Programmer Should Know About Memory
  RedHat Inc.,2007

- Bruce Jacob:
  The Memory System
  Morgan & Claypool Publishers, 2009 (77 pages)

- Francesc Alted
  Why Modern CPUs Are Starving and What Can Be Done about It
  Computing in Science and Engineering, March 2010

# Exercises

- Exercises 1 – 4: **Experimenting**
    - Just play around with different equations and settings to understand your hardware. Try to predict the outcome, before you measure the performance.
    - If possible, try to experiment with your own hardware and compare the results.
      You only need **`numpy`**, **`numexpr`**, **`numba`** and the **`multiprocessing`** module.
    - when you increase the data-set: keep our computer's memory-limit in mind!
    - Discuss the different results within the group.
- Exercises 5 – 6: **Problem solving**
    - time consuming, especially #6 is probably more something for a rainy week-end
    - do only what you think is fun, check otherwise the solutions.

# Acknowledgment

Based on the slides of Francesc Alted

# Extra

# Some High Performance Libraries

- **BLAS**: Routines that provide standard building blocks for performing basic vector and matrix operations.

- **ATLAS**: Memory efficient algorithms as well as SIMD algorithms so as to provide an efficient BLAS implementation.

- **MKL**: (Intel's Math Kernel Library): Like ATLAS, but with support for multi-core and fine-tuned for Intel architecture. Its VML subset computes basic math functions (sin, cos, exp, log...) very efficiently.

- **Numexpr**: Performs relatively simple operations with NumPy arrays without the overhead of temporaries. Can make use of multi-cores.

- **Numba**: Can compile potentially complex Python code involving NumPy arrays.

- **JAX**: high-performance numerical computing (with NumPy arrays): differentiate, vectorize, parallelize, also just in time compilation with GPU support.