The Starving CPU Problem

Or Why Should I Care About Memory Access?

Francesc Alted

Software Architect
Continuum Analytics

Outline

Computing a Polynomial

We want to compute the next polynomial:

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

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

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

Computing a Polynomial

We want to compute the next polynomial:

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

in the range [-1, 1], with a granularity of 10 million points the x axis
...and 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:

Computing $y = 0.25x^3 + 0.75x^2 - 1.5x - 2$ with 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 0.86 sec on our machine (Intel Core i5-3380M CPU @ 2.90GHz). 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 (so as to bypass the GIL). See `poly-mp.py` script.
- Using 2 cores, the 0.86 sec is **slowed down** to 0.88 sec! WTF?
- Can we continue to believe in parallelism at all?

Another (Much Easier) Approach: Factorize

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

Advice

Give optimization a chance before parallelizing!

Another (Much Easier) Approach: Factorize

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

Advice

Give optimization a chance before parallelizing!

Numexpr Can Compute Expressions Way Faster

Numexpr is a JIT compiler, based on NumPy, that optimizes the evaluation of complex expressions. Its use is easy:

Computing $y = 0.25x^3 + 0.75x^2 - 1.5x - 2$ with numexpr

```
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')
```

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 accepts using several processors:

Asking numexpr to use a certain number of threads

```
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 (0.46 sec).

Summary and Open Questions

	1 core	2 core	Parallel Speed-up
NumPy (I)	0.867	0.887	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
C(II)	0.013	0.013	1.00x

- 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?

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*x - 2
```

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

One (Important) Remaining Question

Why numexpr can execute this expression:

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

more than 2x faster than NumPy, even using a single core?

Short answer

By making a more efficient use of the memory resource

.

Outline

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

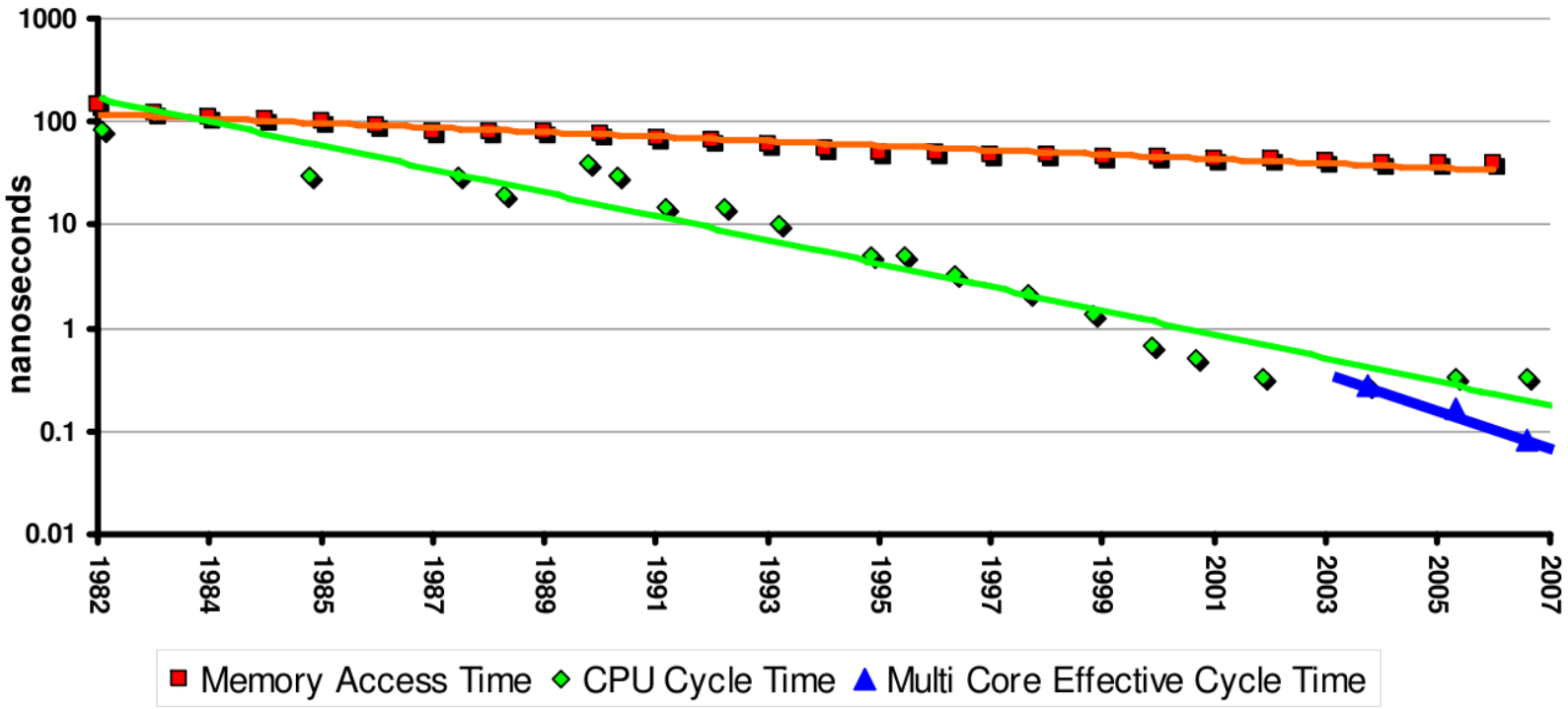
Quote Back in 1996

"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."

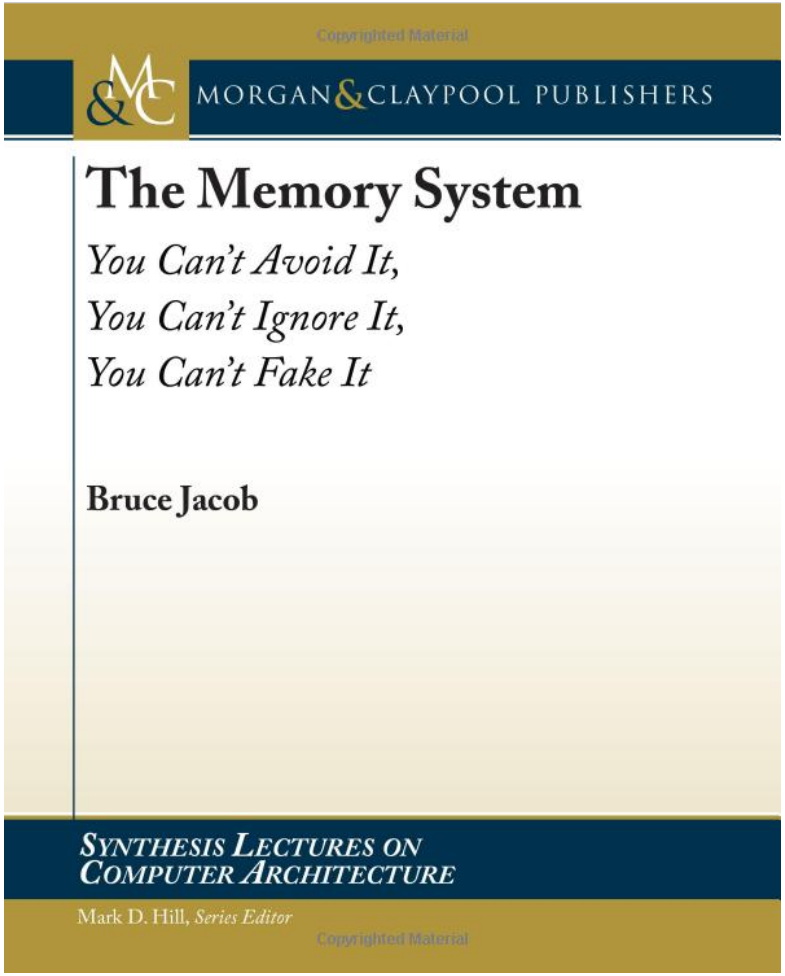
"Over the coming decade, memory subsystem design will be the only important design issue for microprocessors."

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

CPU vs Memory Cycle Trend



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

Known facts (in 2013):

- Memory latency is much slower (between 250x and 500x) than processors and has been an essential bottleneck for the past fifteen years.
- 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.*

Outline

What Is the Industry Doing to Alleviate CPU Starvation?

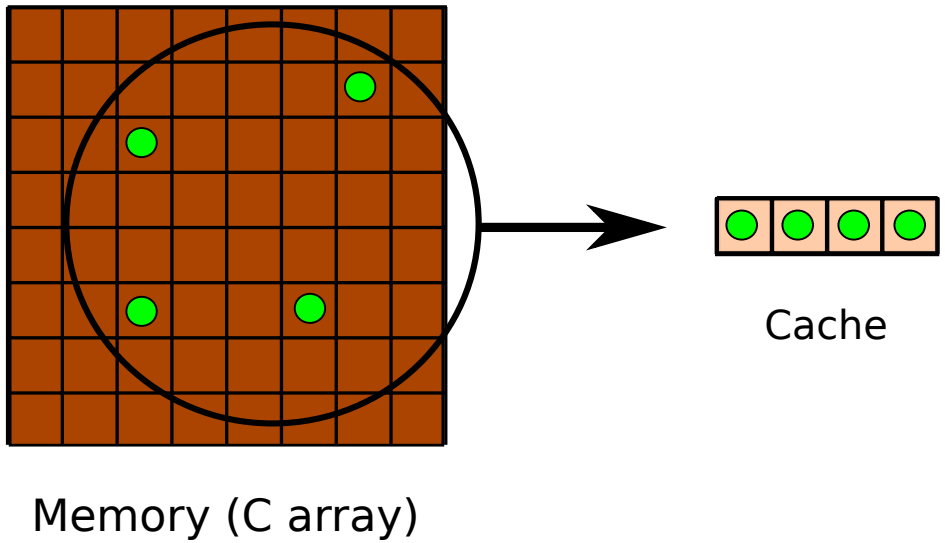
- They are improving memory throughput: cheaper to implement (more data is transmitted on each clock cycle).
- They are adding big caches in the CPU dies.

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.

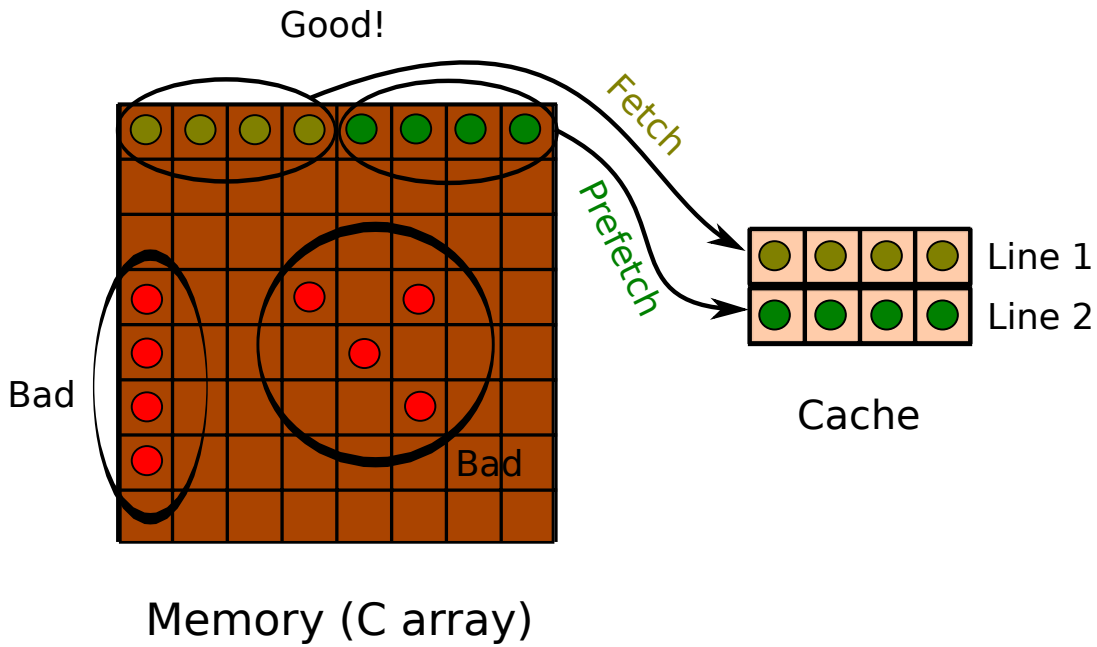
Time Locality

Parts of the dataset are reused



Spatial Locality

Dataset is accessed sequentially

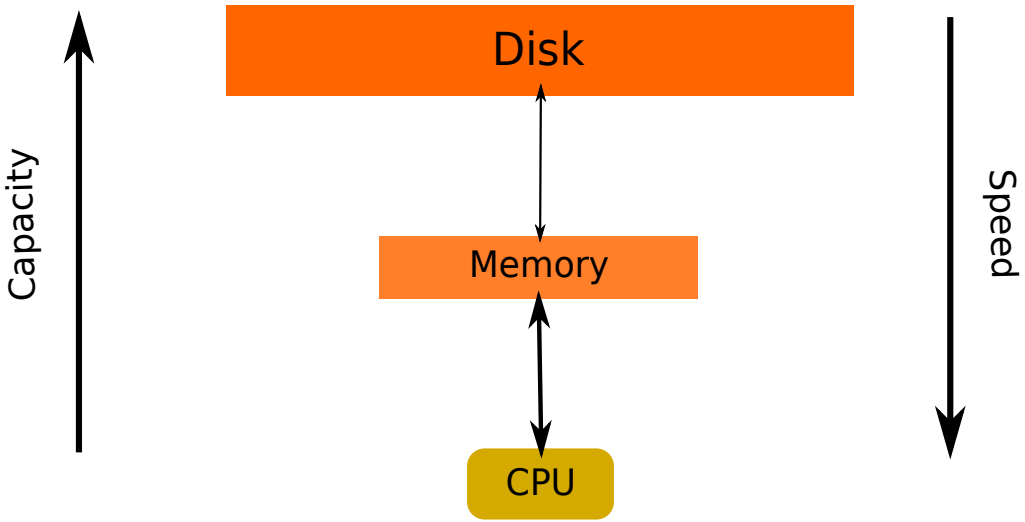


The Hierarchical Memory Model

- 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.

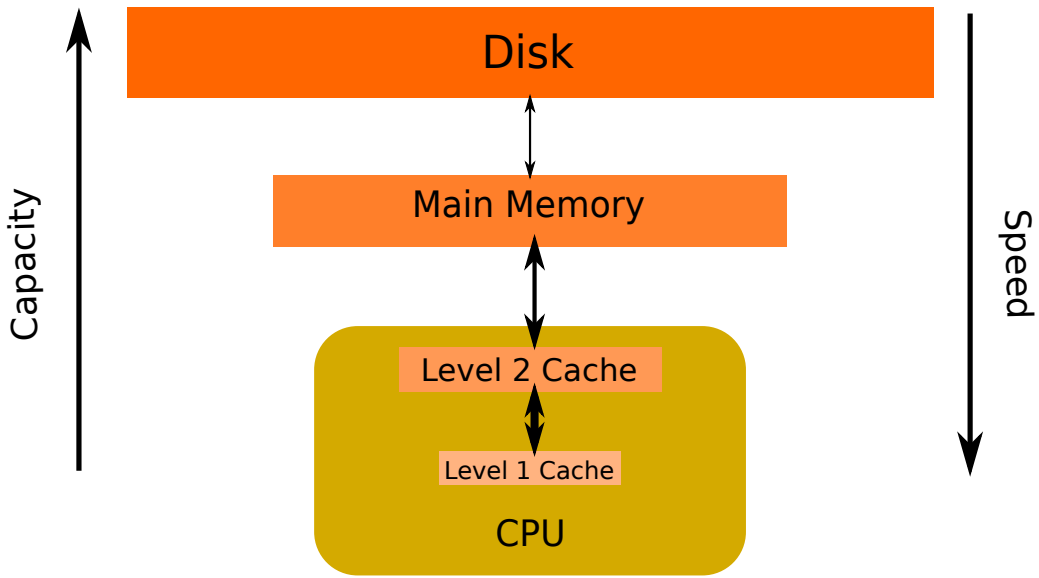
The Primordial Hierarchical Memory Model

Two level hierarchy



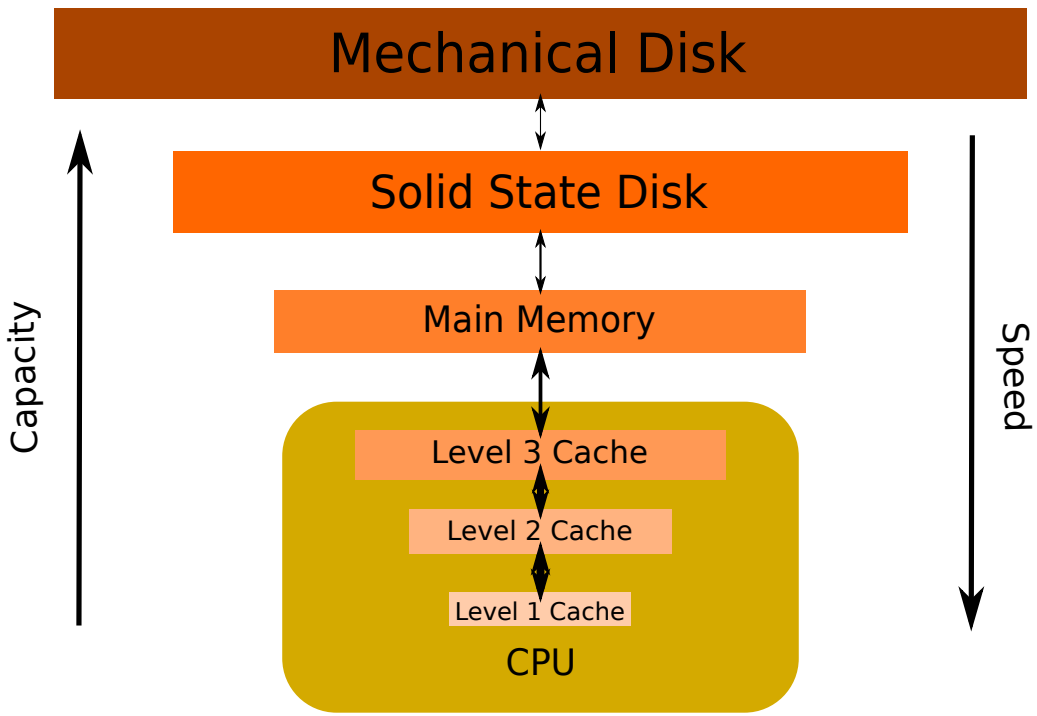
The 2000's Hierarchical Memory Model

Four level hierarchy



The Current Hierarchical Memory Model

Six level (or more) hierarchy



Outline

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.
- 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.

The BIG difference is...

. . . learning assembly language is relatively easy, but understanding how the hierarchical memory model works requires a considerable amount of experience (it's almost more an art than a science!)

Nowadays...

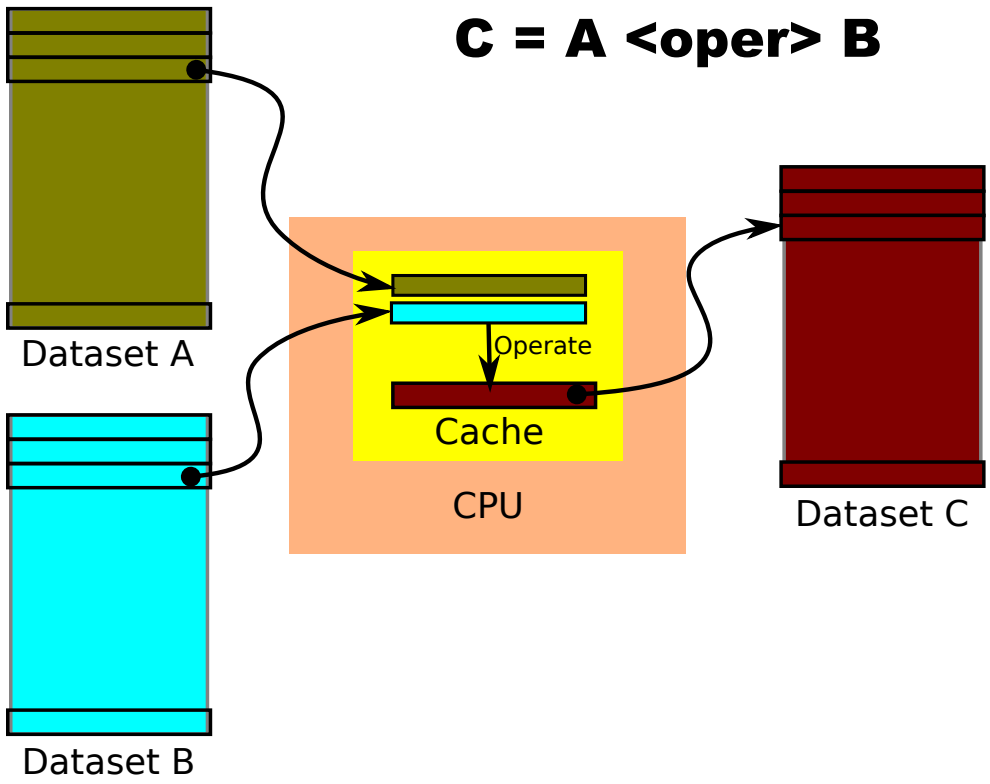
- Every computer scientist must acquire a good knowledge of the hierarchical memory model (and its implications) if they want their applications to run at a decent speed (i.e. they do not want their CPUs to starve too much).
- Memory organization has become now the key factor for optimizing.

The BIG difference is...

...learning assembly language is relatively easy, but understanding how the hierarchical memory model works requires a considerable amount of experience (it's almost more an art than a science!)

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
```

Remember:

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

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
```

Remember:

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

Vectorize Your Code

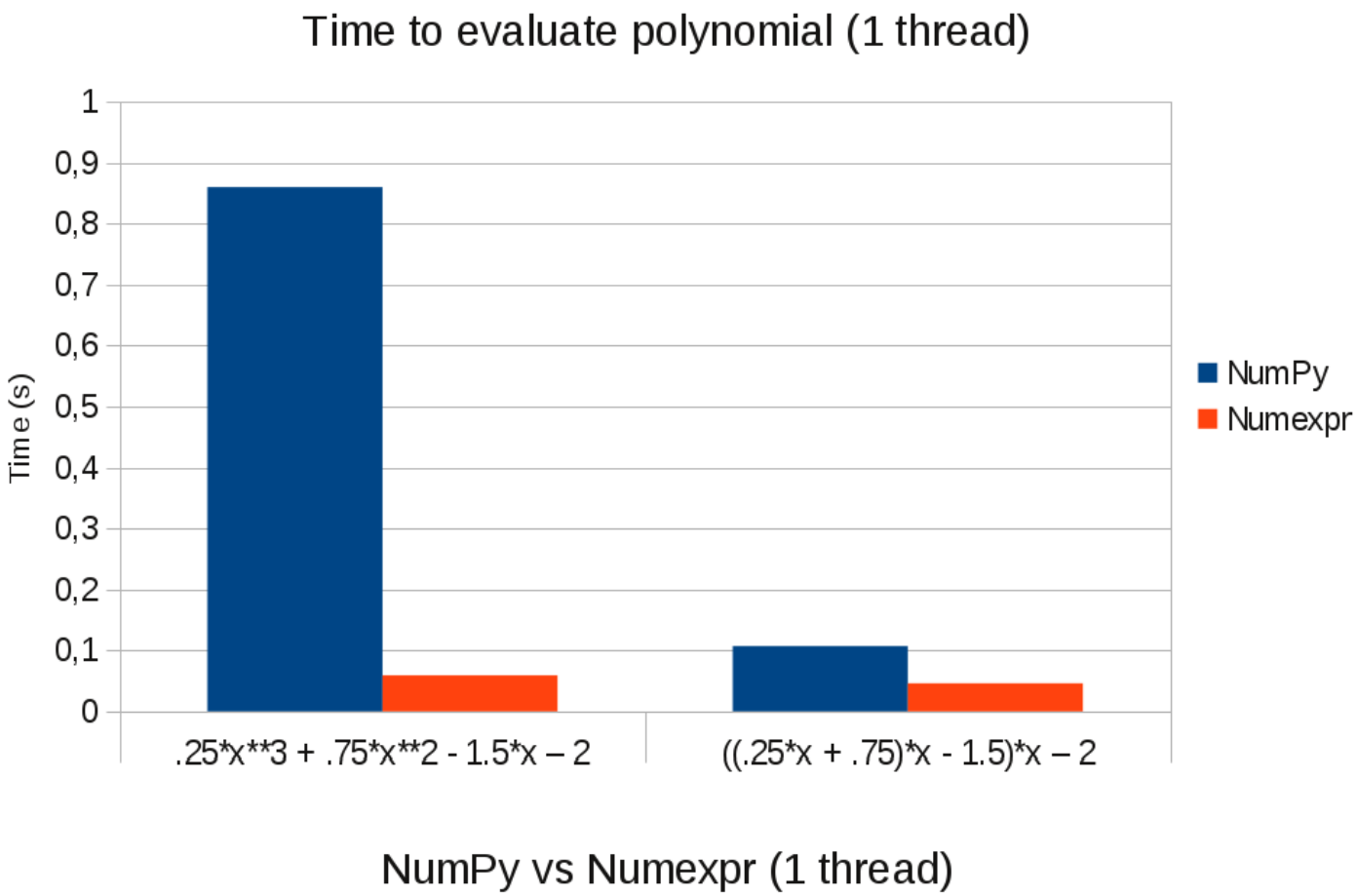
Naive matrix-matrix multiplication: 1264 s (1000x1000 doubles)

```
def dot_naive(a,b):      #  1.5 MFlops
    c = np.zeros((nrows, ncols), dtype='f8')
    for row in xrange(nrows):
        for col in xrange(ncols):
            for i in xrange(nrows):
                c[row,col] += a[row,i] * b[i,col]
    return c
```

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

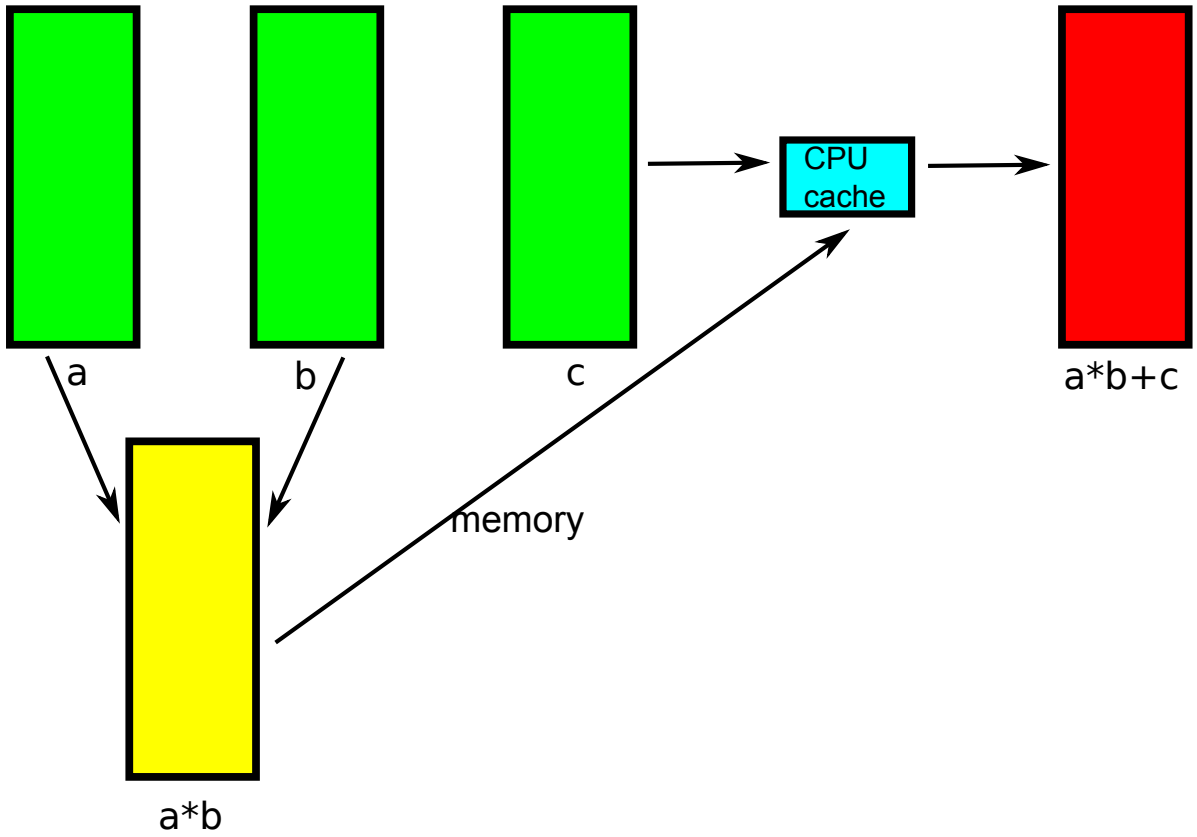
```
def dot(a,b):            #  100 MFlops
    c = np.empty((nrows, ncols), dtype='f8')
    for row in xrange(nrows):
        for col in xrange(ncols):
            c[row, col] = np.sum(a[row] * b[:,col])
    return c
```

Interlude: Resolving More Open Questions



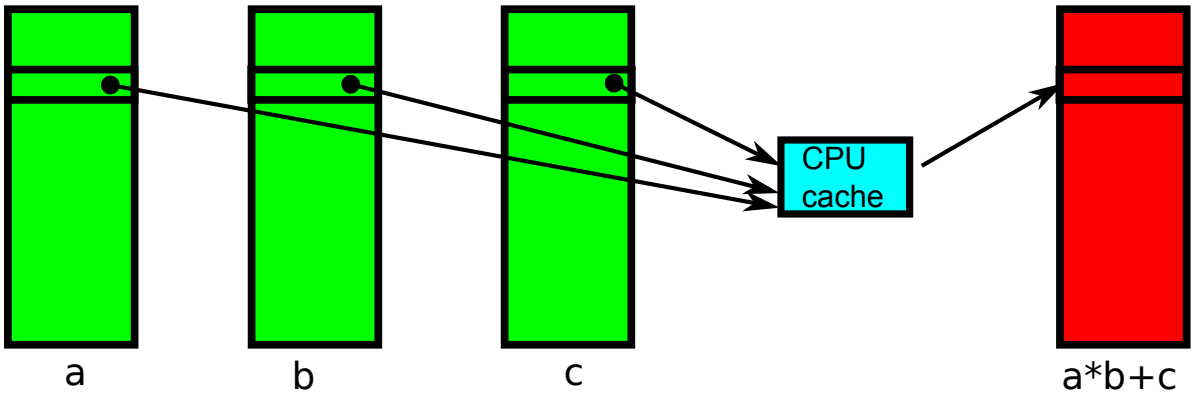
NumPy And Temporaries

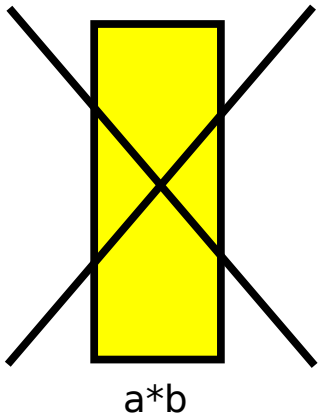
Computing "a*b+c" with NumPy. Temporaries goes to memory.



Numexpr Avoids (Big) Temporaries

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





memory

Some More Mysteries Solved Now!

	1 core	2 core	Parallel Speed-up
NumPy (I)	0.867	0.887	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
C(II)	0.013	0.013	1.0x

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 via LLVM infraestructure.

BLAS/ATLAS/Intel's MKL

Optimizing Memory Access

Using integrated BLAS: 5.6 s (3.5x faster than vectorized)

```
numpy.dot(a,b)     # 350 MFlops
```

Using ATLAS: 0.19s (35x faster than integrated BLAS)

```
numpy.dot(a,b)     # 10 GFlops
```

Using Intel's MKL: 0.11 s (70% faster than LAPACK)

```
numpy.dot(a,b)     # 17 GFlops (2x12=24 GFlops peak)
```

Numexpr: Dealing with Complex Expressions

Numexpr is a specialized virtual machine for evaluating expressions. It accelerates computations by using blocking and by avoiding temporaries.
For example, if "a" and "b" are vectors with 1 million entries each:

Using plain NumPy

```
a**2 + b**2 + 2*a*b     # takes 21.5 ms
```

Using Numexpr (one thread): more than 2x faster!

```
numexpr.evaluate('a**2 + b**2 + 2*a*b')  # 10.2 ms
```

Numexpr: Multithreading Support

Numexpr can use multiple threads easily:

```
numexpr.set_num_threads(8)   # use 8 threads
numexpr.evaluate('a**2 + b**2 + 2*a*b')
 # 3.18 ms, 7x faster than NumPy
```

```
numexpr.set_num_threads(16)   # use 16 threads
numexpr.evaluate('a**2 + b**2 + 2*a*b')
 # 1.98 ms, 11x faster than NumPy
```

Important

Numexpr also has support for Intel's VML (Vector Math Library), so you can accelerate the evaluation of transcendental (sin, cos, atanh, sqrt...) functions too.

Numexpr: Multithreading Support

Numexpr can use multiple threads easily:

```
numexpr.set_num_threads(8)   # use 8 threads
numexpr.evaluate('a**2 + b**2 + 2*a*b')
 # 3.18 ms, 7x faster than NumPy
```

```
numexpr.set_num_threads(16)   # use 16 threads
numexpr.evaluate('a**2 + b**2 + 2*a*b')
 # 1.98 ms, 11x faster than NumPy
```

Important

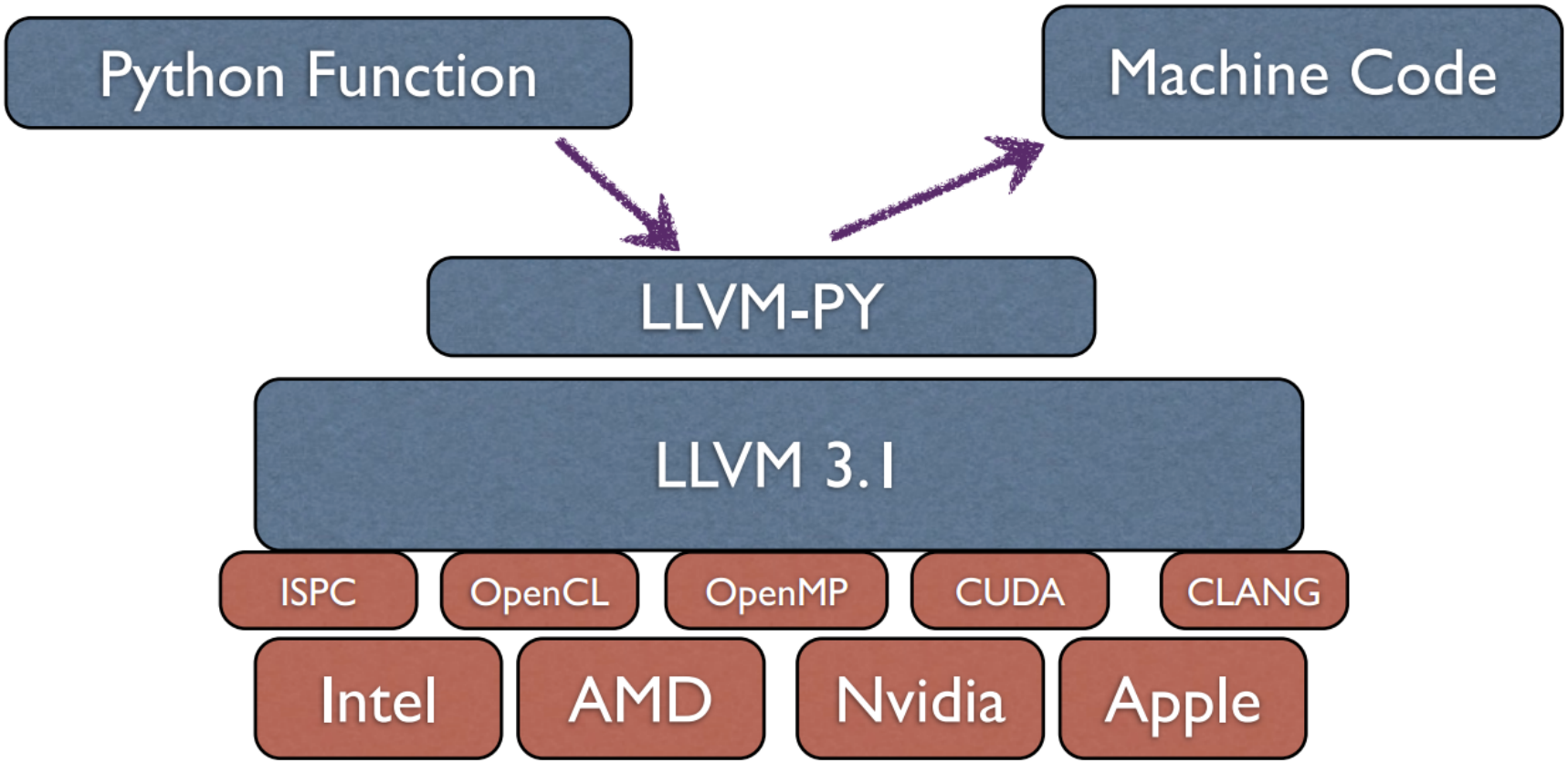
Numexpr also has support for Intel's VML (Vector Math Library), so you can accelerate the evaluation of transcendental (sin, cos, atanh, sqrt...) functions too.

Numba: Overcoming *numexpr* Limitations

NumPy aware dynamic Python compiler using LLVM

- Numba is a JIT that can translate a subset of the Python language into machine code
- It uses LLVM infrastructure behind the scenes
- 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 (MIT-like license).

How Numba Works



Numba Example: Computing the Polynomial

```
from numba import autojit
import numpy as np
N = 10*1000*1000
x = np.linspace(-1, 1, N)
y = np.empty(N, dtype=np.float64)
@autojit
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 core	Parallel Speed-up
NumPy (I)	0.860	0.710	1.2x
NumPy(II)	0.107	0.307	0.35x
Numexpr(I)	0.061	0.036	1.7x
Numexpr(II)	0.048	0.032	1.5x
C(II)	0.013	0.013	1.0x
Numba (I)	0.731	-	-
Numba (II)	0.037	-	-

Compilation time for Numba: 0.321 sec

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, so as to bypass the GIL).
  - 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!

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, so as to bypass the GIL).
  - 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.
- Leverage existing memory-efficient libraries for performing 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

What's Next

In the following exercises we will:

- Experiment with different libraries so as to check their advantages and limitations.
- Learn about under which situations you can expect your problems to scale (and when not!).
- Answer remaining pending open questions.

Questions?

Contact:

francesc@continuum.io

The 'Big Iron' Machine for Exercises

4 blades like this one: