When Parallelization Does Not Help

The Starving CPU Problem

Francesc Alted

Freelance Developer and PyTables Creator

Advanced Scientific Programming in Python
2010 Autumn School, Trento, Italy

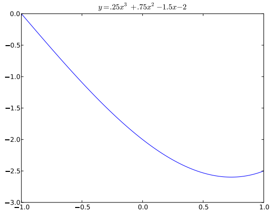
Outline

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^{-7} in the x axis



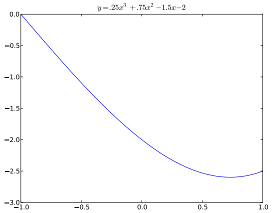
...and want to do that as FAST as possible...

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^{-7} in 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 1.58 sec on our machine (Intel Xeon E5520 @ 2.27GHz). 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).
- Using 2 cores, the 1.58 sec is reduced down to 1.13 sec, which is a 1.4x improvement. Pretty good!
- We are done! Or perhaps not?

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 1.58 sec to 0.28 sec, which is considerably faster than using two processors with the initial approach (1.13 sec).

Advice

Give a chance to optimization 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 1.58 sec to 0.28 sec, which is considerably faster than using two processors with the initial approach (1.13 sec).

Advice

Give a chance to optimization 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 0.13 sec to complete, which is 12x faster than the original NumPy expression (1.58 sec).

Fine-tune Expressions with Numexpr

- Numexpr is also sensible (but not really too much) to computer-friendly expressions like:
  (II) `y = ((.25*x + .75)*x - 1.5)*x - 2`
- Numexpr takes 0.12 sec for the above (0.13 sec were needed for the original expression, that's a 1.1x faster)

Summary and Open Questions

	1 core	2 core	Parallel Speed-up
NumPy (I)	1.58	1.3	1.4x
NumPy(II)	0.28	0.55	0.5x
Numexpr(I)	0.13	0.075	1.7x
Numexpr(II)	0.12	0.069	1.7x
C(I)	1.25	-	-
C(II)	0.048	-	-

- If all the approaches perform the same computations, all in C space, why the large differences in performance?
- Why the different approaches do not scale similarly in parallel mode?

Outline

Once Upon A Time...

- In the 1970s and 1980s the memory subsystem was able to deliver all the data that processors required in time.
- In the good old days, the processor was the key bottleneck.
- But in the 1990s things started to change...

Once Upon A Time...

- In the 1970s and 1980s the memory subsystem was able to deliver all the data that processors required in time.
- In the good old days, the processor was the key bottleneck.
- But in the 1990s things started to change...

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*

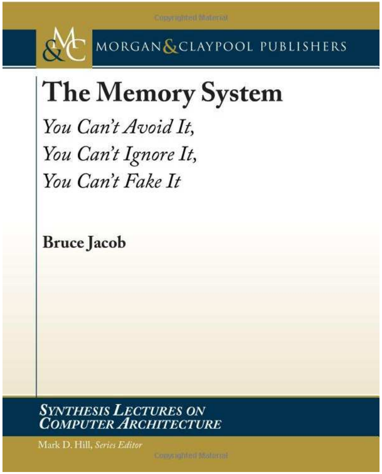
Quote Back in 1996

"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 CPU Starvation Problem

Known facts (in 2010):

- Memory latency is much higher (around 250x) 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 (about 25x).

The result is that CPUs in our current computers are suffering from a serious data starvation problem: *they could consume (much!) more data than the system can possibly deliver.*

Outline

What Is the Industry Doing to Alleviate CPU Starvation?

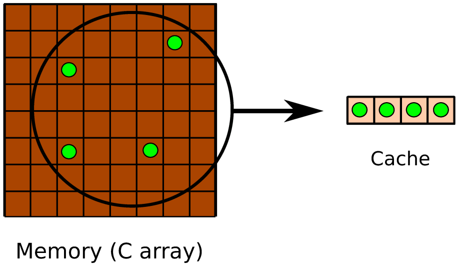
- They are improving memory throughput: cheap to implement (more data is transmitted on each clock cycle).
- They are adding big caches in the CPU dies.

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.

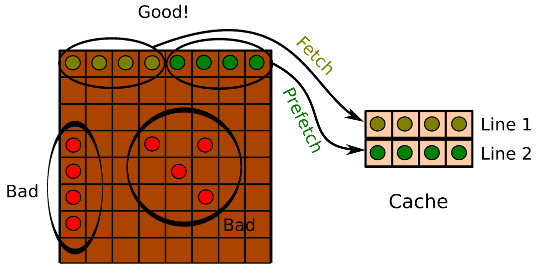
Time Locality

Parts of the dataset are reused



Spatial Locality

Dataset is accessed sequentially



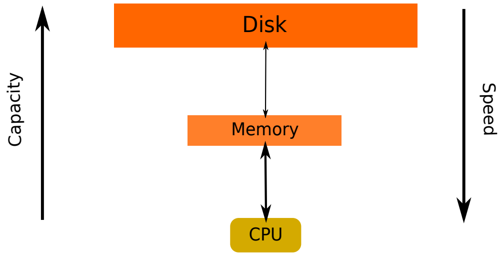
Memory (C array)

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.

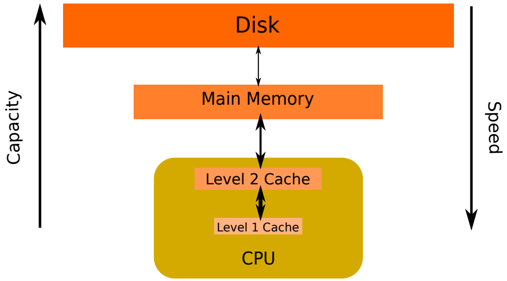
The Primordial Hierarchical Memory Model

Two level hierarchy



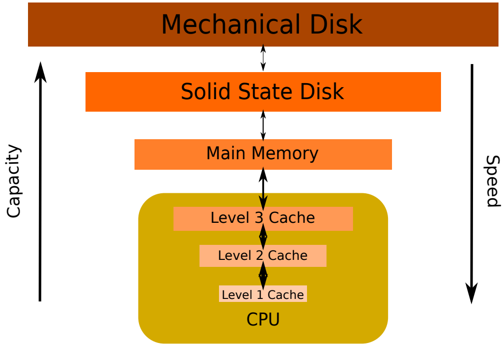
The Current Hierarchical Memory Model

Four level hierarchy



The Forthcoming Hierarchical Memory Model

Six level (or more) hierarchy



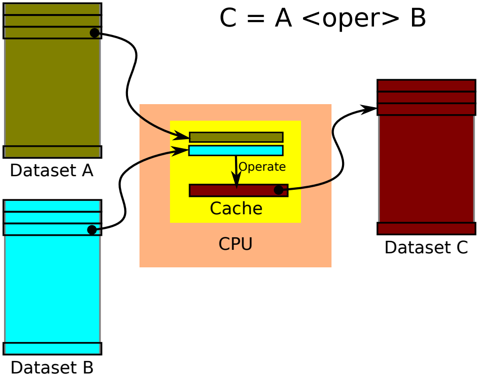
You Can't Ignore The Memory Subsystem

- 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 now become the key factor for optimizing.

Outline

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)

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)

Vectorize Your Code

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

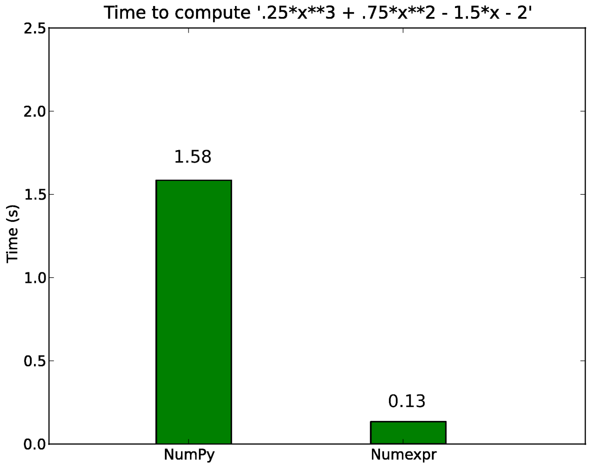
```
def dot_naive(a,b):      #  1.5 MFlops
    c = np.zeros((nrows, ncols), dtype='f8')
    for row in xrange(nrows):
        for col in xrange(ncols):
            for i in xrange(nrows):
                c[row,col] += a[row,i] * b[i,col]
    return c
```

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
```

Outline

Results For Polynomial Computation (I)

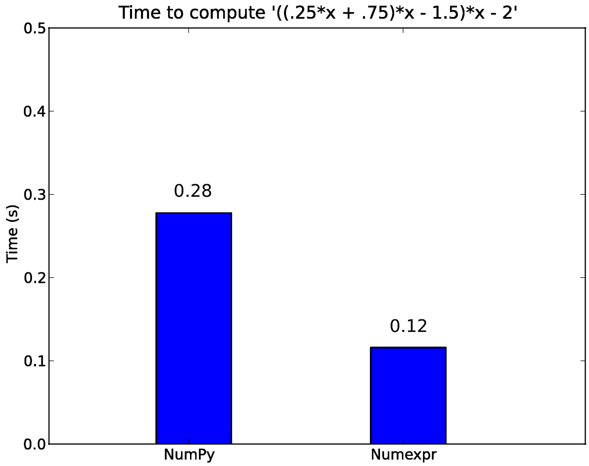


Numexpr Working As A JIT Compiler

	NumPy	Numexpr
x**3	pow(x,3)	x*x*x
time	23.6 ms	4.32 ms

Numexpr can optimize more scenarios than NumPy can

Results For In-Memory Computation (II)

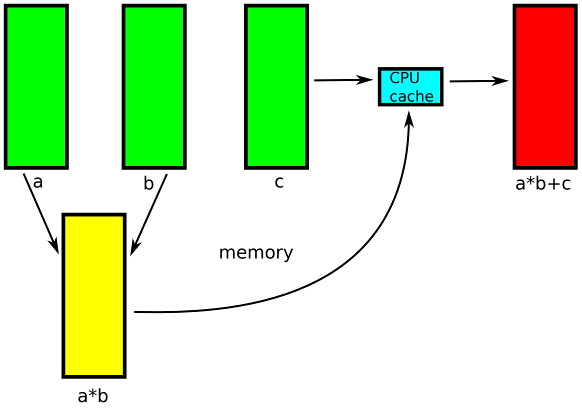


Numexpr Is Aware of Memory Access

- In this case, the number of low-level operations performed by NumPy and Numexpr are exactly the same.
- The only reason that can account for the 2x difference in speed is how memory is accessed (Numexpr is applying the blocking technique).

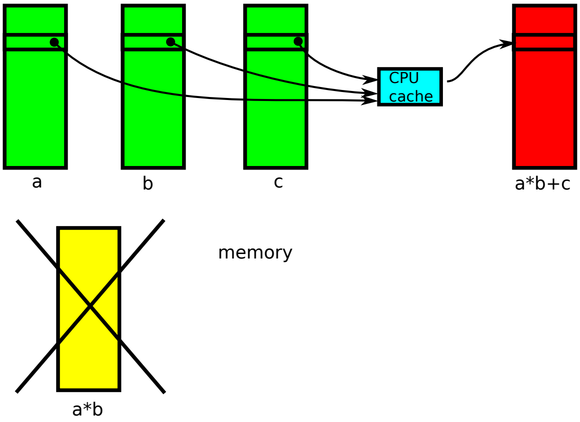
NumPy And Temporaries

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.



Outline

Why High Performance Libraries?

- High perfomance libraries are made by people that knows very well the different optimization techniques.
- You may be tempted to create original algorithms that can be faster than these, but in general, it is very difficult to beat them.
- In some cases, it may take some time to get used to them, but the effort pays off in the long run.

Outline

Some High Performance Libraries

MKL (Intel's Math Kernel Library): Uses memory efficient algorithms as well as SIMD and multi-core algorithms → linear algebra operations.

VML (Intel's Vector Math Library): Uses SIMD and multi-core to compute basic math functions (sin, cos, exp, log...) in vectors.

Numexpr: Performs potentially complex operations with NumPy arrays without the overhead of temporaries. Can make use of multi-cores.

Blosc: A multi-threaded compressor that can transmit data from caches to memory, and back, at speeds that can be larger than a OS `memcpy()`.

PyTables: Can combine all of the above (except MKL) for performing optimal out-of-core computations on arbitrarily large datasets.

ATLAS/Intel's MKL: Optimize 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 ATLAS)

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

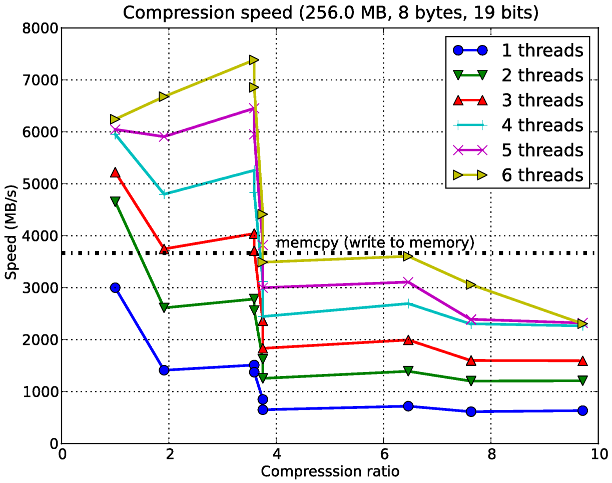
Numexpr: Dealing with Complex Expressions

- Wears a specialized virtual machine for evaluating expressions.
- It accelerates computations by using blocking and by avoiding temporaries.
- Multi-threaded: can use several cores automatically.
- It has support for Intel's VML (Vector Math Library), so you can accelerate the evaluation of transcendental (sin, cos, atanh, sqrt. . .) functions too.

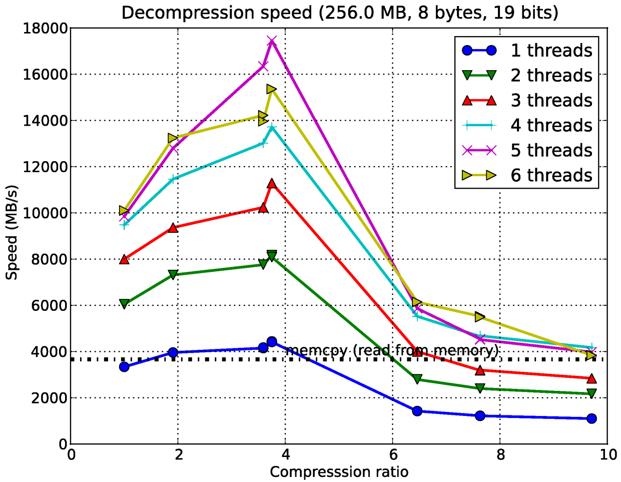
Blosc: A Blocked, Shuffling and Loss-Less Compression Library

- Blosc is a new, loss-less compressor for binary data. It's optimized for speed, not for high compression ratios.
- It is based on the FastLZ compressor, but with some additional tweaking:
    - It works by splitting the input dataset into blocks that fit well into the level 1 cache of modern processors.
    - It can shuffle bytes very efficiently for improved compression ratios (using the data type size meta-information).
    - Makes use of SSE2 vector instructions (if available).
    - Multi-threaded (via pthreads).
- Free software (MIT license).

Blosc: Beyond `memcpy()` Performance (I)



Blosc: Beyond `memcpy()` Performance (II)



tables.Expr: Operating With Very Large Arrays

- `tables.Expr` is an optimized evaluator for expressions of disk-based arrays.
- It is a combination of the Numexpr advanced computing capabilities with the high I/O performance of PyTables.
- It is similar to `numpy.memmap`, but with important improvements:
  - Deals transparently (and efficiently!) with temporaries.
  - Works with arbitrarily large arrays, no matter how much virtual memory is available, what version of OS version you're running (works with both 32-bit and 64-bit OS's), which Python version you're using (2.4 and higher), or what's the phase of the moon.
  - Can deal with compressed arrays seamlessly.

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

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 you:

- Will learn how to optimize the evaluation of arbitrarily complex expressions.
- Will experiment with in-memory and out-of-memory computation paradigms.
- Will check how compression can be useful in out-of-memory calculations (and maybe in some in-memory ones too!).

Questions?

Contact:

faltet@pytables.org