Memory-Efficient Computing

(Or How To Make Your Programs Faster Without Parallelizing)

Francesc Alted

Freelance Developer and PyTables Creator

Advanced Scientific Programming in Python
February 8th-12th, 2010. Warsaw - Poland

Outline

1. Motivation
2. The Data Access Issue
    - A Bit of (Personal) Computing History
    - CPU Starvation and The Hierarchical Memory Model
    - Techniques For Fighting CPU Starvation
3. The Role Of Compression In Data Access
    - Eliminating Data Redundancy
    - Blosc: A BLOcking Shuffler & Compressor
    - An Application of Blosc: the PyTables Database

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 1.20 sec on our machine (Intel Core2 @ 2.4 GHz). 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 1.20 sec is reduced down to 0.76 sec, which is a 1.6x 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 1.20 sec to 0.50 sec, which is considerably faster than using two processors with the initial approach (0.76 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 1.20 sec to 0.50 sec, which is considerably faster than using two processors with the initial approach (0.76 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 0.24 sec to complete, which is 5x faster than the original NumPy expression (1.20 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 0.19 sec for the above (0.24 sec were needed for the original expression, that's a 1.25x faster)

Summary and Open Questions

	1 core	2 core	Parallel Speed-up
NumPy (I)	1.20	0.76	1.6x
NumPy(II)	0.50	0.39	1.3x
Numexpr(I)	0.24	0.14	1.7x
Numexpr(II)	0.19	0.12	1.7x
C(II)	0.075	0.055	1.4x

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

Outline

1. Motivation

2. The Data Access Issue
   - A Bit of (Personal) Computing History
   - CPU Starvation and The Hierarchical Memory Model
   - Techniques For Fighting CPU Starvation

3. The Role Of Compression In Data Access
   - Eliminating Data Redundancy
   - Blosc: A BLOcking Shuffler & Compressor
   - An Application of Blosc: the PyTables Database

First Commodity Processors
(Early and middle 1980s)

- Processors and memory evolved more or less in step.
- Memory clock access in early 1980s was at ~ 1MHz, the same speed than CPUs.

Intel 8086, 80286 and i386
(Middle and late 1980's)

- Memory still pretty well matched CPU speed.
- The 16MHz i386 came out; memory still could keep up with it.

Intel i486 and AMD Am486
(Early 1990s)

- Increases in memory speed started to stagnate, while CPU clock rates continued to skyrocket to 100 MHz and beyond.
- In a single-clock, a 100 MHz processor consumes a word from memory every 10 nsec. This rate is impossible to sustain even with *present-day* RAM.
- The first on-chip cache appeared (8 KB for i486 and 16 Kb for i486 DX).

Intel Pentium and AMD K5/K6
(Middle and late 1990s)

- Processor speeds reached unparalleled extremes, before hitting the magic 1 GHz figure.
- A huge abyss opened between the processors and the memory subsystem: up to 50 wait states for each memory read or write.

Intel Pentium 4 and AMD Athlon
(Early and middle 2000s)

- The strong competition between Intel and AMD continued to drive CPU clock cycles faster and faster (up to 0.25 ns, or 4 GHz).
- The increased impedance mismatch with memory speeds brought about the introduction of a second level cache.

Intel Core2 and AMD Athlon X2

(Middle 2000s)

- The size of integrated caches is getting really huge (up to 12 MB).
- Chip makers realized that they can't keep raising the frequency forever → enter the multi-core age.
- Users start to scratch their heads, wondering how to take advantage of multi-core machines.

Intel Core2 and AMD Athlon X2
(Middle 2000s)

- The size of integrated caches is getting really huge (up to 12 MB).
- Chip makers realized that they can't keep raising the frequency forever $\rightarrow$ enter the multi-core age.
- Users start to scratch their heads, wondering how to take advantage of multi-core machines.

Intel Core i7 and AMD Phenom (Late 2000s)

- 4-core on-chip CPUs become the most common configuration.
- 3-levels of on-chip cache is the standard now.

Where We Are now (2010)

- Memory latency is much slower (around 150x) 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 lagging behind processors (about 25x slower).
- In order to achieve better performance, CPU makers are implementing additional levels of caches, as well as increasing cache size.
- Recently, CPU speeds have stalled as well, limited now by power dissipation problems. So, in order to be able to offer more speed, CPU vendors are packaging several processors (cores) in the same die.

Outline

1. Motivation
2. The Data Access Issue
   - A Bit of (Personal) Computing History
   - CPU Starvation and The Hierarchical Memory Model
   - Techniques For Fighting CPU Starvation
3. The Role Of Compression In Data Access
   - Eliminating Data Redundancy
   - Blosc: A BLOcking Shuffler & Compressor
   - An Application of Blosc: the PyTables Database

The CPU Starvation Problem

- Over the last 25 years CPUs have undergone an exponential improvement on their ability to perform massive numbers of calculations extremely quickly.
- However, the memory subsystem hasn't kept up with CPU evolution.
- 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.*

Can't Memory Latency Be Reduced to Keep Up with CPUs?

- To improve latency figures, we would need:
  - more wire layers
  - more complex ancillary logic
  - more frequency (and voltage):

    $$Energy = Capacity \times Voltage^2 \times Frequency$$

- That's too expensive for commodity SDRAM.

What Is the Industry Doing to Alleviate CPU Starvation?

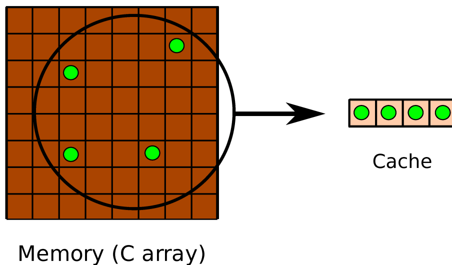
- They are improving memory throughput: cheap to implement (more data is transmitted on each clock cycle).
- They are adding big caches in the CPU dies.

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.

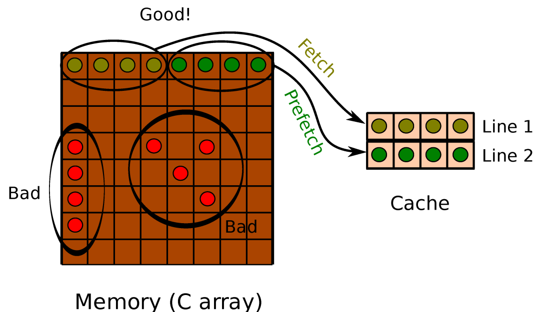
Time Locality

Parts of the dataset are reused



Spatial Locality

Dataset is accessed sequentially

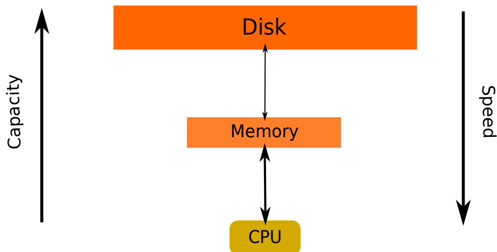


The Hierarchical Memory Model

- 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.

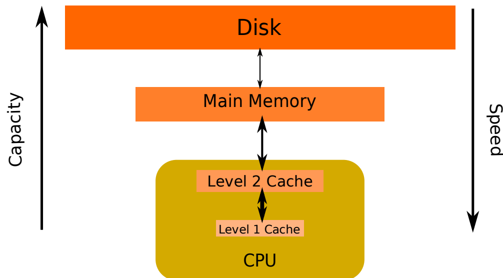
The Primordial Hierarchical Memory Model

Two level hierarchy



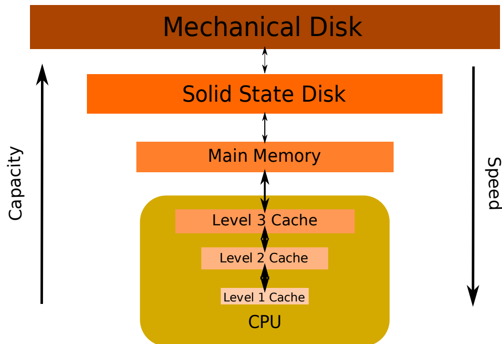
The Current Hierarchical Memory Model

Four level hierarchy



The Forthcoming Hierarchical Memory Model

Six level (or more) hierarchy



Outline

1. Motivation
2. The Data Access Issue
   - A Bit of (Personal) Computing History
   - CPU Starvation and The Hierarchical Memory Model
   - Techniques For Fighting CPU Starvation
3. The Role Of Compression In Data Access
   - Eliminating Data Redundancy
   - Blosc: A BLOcking Shuffler & Compressor
   - An Application of Blosc: the PyTables Database

Once Upon A Time...

- In the 1970s and 1980s many computer 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...

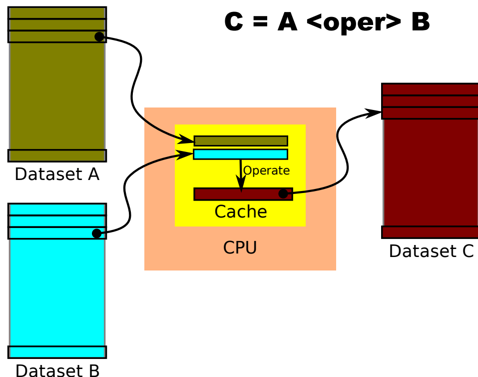
- Every computer scientist must acquire a good knowledge of the hierarchical memory model (and its implications) if they want their applications to run at a decent speed (i.e. they do not want their CPUs to starve too much).
- Memory organization has become now the key factor for optimizing.

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 I

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:



The Blocking Technique II

- This is not new at all: it has been in use for out-of-core computations since the dawn of computers.
- However, the meaning of *out-of-core* is changing, since the *core* does not refer to the main memory anymore: it now means something more like *out-of-cache*.
- Although this technique is easy to apply in some cases (e.g. element-wise array computations), it can be potentially difficult to *efficiently* implement in others.

Good News!

Fortunately, many useful algorithms using blocking have been developed by others that you can use ☺

The Blocking Technique II

- This is not new at all: it has been in use for out-of-core computations since the dawn of computers.
- However, the meaning of *out-of-core* is changing, since the *core* does not refer to the main memory anymore: it now means something more like *out-of-cache*.
- Although this technique is easy to apply in some cases (e.g. element-wise array computations), it can be potentially difficult to *efficiently* implement in others.

Good News!

Fortunately, many useful algorithms using blocking have been developed by others that you can use ☺

I'm A Python Guy, What Should I Do?

- Can we, Python users, get maximum efficiency for our programs? Simply put: Yes, we can.
- To do this, it is a good idea to *become familiar with all the weaponry* that Python and third-party packages offer: many fine tools are waiting for you.
- Then, you should take some time to *think* and try to *express your situation in terms of the problems these weapons are designed to attack*.
- It is not generally possible to make your problem disappear with a single swipe: *this is an iterative process* (you never stop learning!)

Use NumPy & Family

- NumPy is the standard package for dealing with multidimensional arrays in a flexible and efficient way.
- There are many packages that add a great deal of functionality to NumPy. Here are some:
  - SciPy (most of you should know what this is)
  - matplotlib (2-D plotter)
  - MayaVi (3-D visualizer)
  - PyTables (fast and easy access to data on-disk)
  - scikits.timeseries (manipulation of time series)
- In case you don't use GNU/Linux, distribution packages like *python(x,y)* and *EPD* make your life (much!) easier.

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
```

Make Use of Optimized Functions and Libraries

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

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

Using Intel's MKL: 0.11 s (50x faster than integrated BLAS)

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

Tip

If you are performing linear algebra calculations, you should try to link NumPy with Atlas, Intel's MKL or similar tools: you will see a significant boost in performance.

Make Use of Optimized Functions and Libraries

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

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

Using Intel's MKL: 0.11 s (50x faster than integrated BLAS)

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

Tip

If you are performing linear algebra calculations, you should try to link NumPy with Atlas, Intel's MKL or similar tools: you will see a significant boost in performance.

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 33.3 ms
```

Using Numexpr: more than 4x faster!

```
numexpr.evaluate('a**2 + b**2 + 2*a*b')  # takes 8.0 ms
```

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: 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 33.3 ms
```

Using Numexpr: more than 4x faster!

```
numexpr.evaluate('a**2 + b**2 + 2*a*b')  # takes 8.0 ms
```

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.

Operating with Very Large Arrays? Use `tables.Expr`

- `tables.Expr` is an optimized evaluator for expressions of disk-based arrays (introduced in PyTables 2.2b1).
- 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 the phase of the moon.
  - Can deal with compressed arrays seamlessly.

An Example of a tables.Expr Calculation

- Let “a” and “b” be single-precision matrices with 1 billion entries (1000x1000000) each (total working set of 11.2 GB, result included).
- Compute the expression “a*b+1” on a 8 GB RAM machine:

Using numpy.memmap

```
r = np.memmap(rfilename, 'float32', 'w+', shape)
for i in xrange(nrows):  # takes 166 s and 11.3 GB
    r[i] = eval('a*b+1', {'a':a[i], 'b':b[i]})
```

Using tables.Expr: 17% faster

```
r = f.createCArray('/', 'r', tb.Float32Atom(), shape)
e = tb.Expr('a*b+1')
e.setOutput(r1)
e.eval()    # takes 141 s and 180 MB
```

An Example of a tables.Expr Calculation

- Let “a” and “b” be single-precision matrices with 1 billion entries (1000x1000000) each (total working set of 11.2 GB, result included).
- Compute the expression “a*b+1” on a 8 GB RAM machine:

Using numpy.memmap

```
r = np.memmap(rfilename, 'float32', 'w+', shape)
for i in xrange(nrows):  # takes 166 s and 11.3 GB
    r[i] = eval('a*b+1', {'a':a[i], 'b':b[i]})
```

Using tables.Expr: 17% faster

```
r = f.createCArray('/', 'r', tb.Float32Atom(), shape)
e = tb.Expr('a*b+1')
e.setOutput(r1)
e.eval()     # takes 141 s and 180 MB
```

tables.Expr and Compression

Activating compression can accelerate out-of-core computations in many cases:

Using zlib (opt. level 1)

takes 78 s: 1.8x faster than with uncompressed arrays.

Using LZO

takes 57 s: 2.5x faster than with uncompressed arrays.

Compression helps to get better performance in disk I/O. No news.

tables.Expr and Compression

Activating compression can accelerate out-of-core computations in many cases:

Using zlib (opt. level 1)

takes 78 s: 1.8x faster than with uncompressed arrays.

Using LZO

takes 57 s: 2.5x faster than with uncompressed arrays.

Compression helps to get better performance in disk I/O. No news.

tables.Expr and Compression

Activating compression can accelerate out-of-core computations in many cases:

Using zlib (opt. level 1)

takes 78 s: 1.8x faster than with uncompressed arrays.

Using LZO

takes 57 s: 2.5x faster than with uncompressed arrays.

Compression helps to get better performance in disk I/O. No news.

Some Words about Multiple Cores and GIL

- The Global Interpreter Lock effectively limits the use of several cores *simultaneously* while in Python space.
- *However*, most of the code out there is limited by memory access, not CPU, so GIL is generally not a problem.
- In addition, many optimized libraries unlock the GIL while doing I/O or performing computations in C space, actually allowing the use of several cores simultaneously.

Advice

Consider making multi-threaded or multi-process programs only as a last resort . You should first see if CPU is the actual problem and then try specialized packages (Atlas, MKL...) first.

Some Words about Multiple Cores and GIL

- The Global Interpreter Lock effectively limits the use of several cores *simultaneously* while in Python space.
- *However*, most of the code out there is limited by memory access, not CPU, so GIL is generally not a problem.
- In addition, many optimized libraries unlock the GIL while doing I/O or performing computations in C space, actually allowing the use of several cores simultaneously.

Advice

Consider making multi-threaded or multi-process programs only as a last resort . You should first see if CPU is the actual problem and then try specialized packages (Atlas, MKL...) first.

Time to Answer Some Pending Questions

Outline

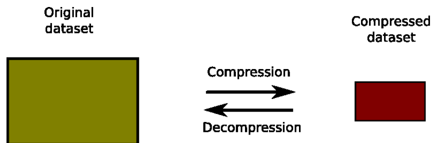
1 Motivation

2 The Data Access Issue
- A Bit of (Personal) Computing History
- CPU Starvation and The Hierarchical Memory Model
- Techniques For Fighting CPU Starvation

3 The Role Of Compression In Data Access
- Eliminating Data Redundancy
- Blosc: A BLOcking Shuffler & Compressor
- An Application of Blosc: the PyTables Database

The Compression Process

- A compression algorithm looks in the dataset for redundancies and dedups them. The usual outcome is a smaller dataset:



The Role of Compression in Data Access

- Compression has *already* helped accelerate reading and writing large datasets from/to disks over the last 10 years.
- It generally takes less time to read/write a small (compressed) dataset than a larger (uncompressed) one, even taking into account the (de-)compression times.

Crazy question:

Given the gap between processors and memory speed, could compression accelerate the transfer from memory to the processor, also?

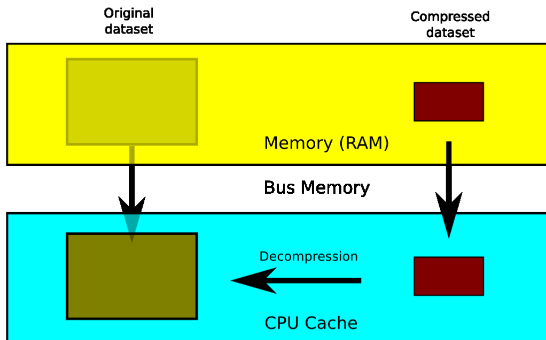
The Role of Compression in Data Access

- Compression has *already* helped accelerate reading and writing large datasets from/to disks over the last 10 years.
- It generally takes less time to read/write a small (compressed) dataset than a larger (uncompressed) one, even taking into account the (de-)compression times.

Crazy question:

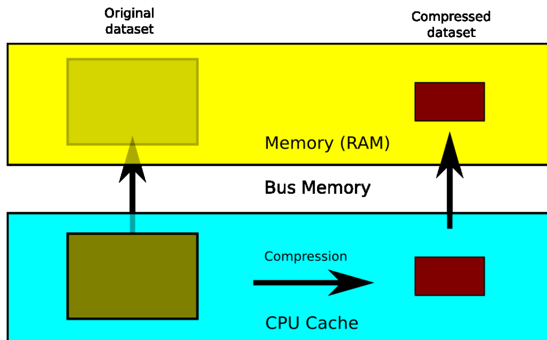
Given the gap between processors and memory speed, could compression accelerate the transfer from memory to the processor, also?

Reading Compressed Datasets



Transmission + decompression processes faster than direct transfer?

Writing Compressed Datasets



Compression + transmission processes faster than direct transfer?

The Challenge: Faster Memory I/O by Using Compression?

What we need:

Extremely fast compressors/decompressors.

What we should renounce:

High compression ratios.

The Challenge: Faster Memory I/O by Using Compression?

What we need:

Extremely fast compressors/decompressors.

What we should renounce:

High compression ratios.

Applications for Fast In-Memory Compression

- We could store more data in a given amount of RAM.
- When a large quantity of data needs to be accessed sequentially, access time could be reduced (if compression is fast enough).

Not good for random access...

To get a single word, you would need to uncompress an entire compressed block.

But...

Many algorithms out there have already been blocked: it should be easy to implement compression for them.

Applications for Fast In-Memory Compression

- We could store more data in a given amount of RAM.
- When a large quantity of data needs to be accessed sequentially, access time could be reduced (if compression is fast enough).

Not good for random access. . .

To get a single word, you would need to uncompress an entire compressed block.

But. . .

Many algorithms out there have already been blocked: it should be easy to implement compression for them.

Applications for Fast In-Memory Compression

- We could store more data in a given amount of RAM.
- When a large quantity of data needs to be accessed sequentially, access time could be reduced (if compression is fast enough).

Not good for random access...

To get a single word, you would need to uncompress an entire compressed block.

But...

Many algorithms out there have already been blocked: it should be easy to implement compression for them.

The Current State of Compressors

- Generally speaking, current compressors do not yet achieve speeds that would allow programs to handle compressed datasets faster than uncompressed data.
- As CPUs have become faster, the trend has been to shoot for high compression ratios, and not so much to reach faster speeds.
- There are some notable exceptions like LZO, LZF and FastLZ, which are very fast compressor/decompressors, but they're still not fast enough to hit our goal.

We need something better!

The Current State of Compressors

- Generally speaking, current compressors do not yet achieve speeds that would allow programs to handle compressed datasets faster than uncompressed data.
- As CPUs have become faster, the trend has been to shoot for high compression ratios, and not so much to reach faster speeds.
- There are some notable exceptions like LZO, LZF and FastLZ, which are very fast compressor/decompressors, but they're still not fast enough to hit our goal.

We need something better!

Outline

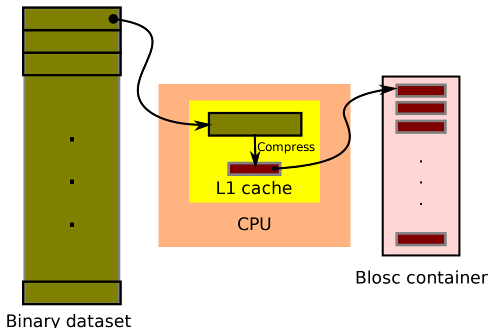
1. Motivation
2. The Data Access Issue
   - A Bit of (Personal) Computing History
   - CPU Starvation and The Hierarchical Memory Model
   - Techniques For Fighting CPU Starvation
3. The Role Of Compression In Data Access
   - Eliminating Data Redundancy
   - Blosc: A BLOcking Shuffler & Compressor
   - An Application of Blosc: the PyTables Database

Blosc: a BLOcking, Shuffling & 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).
- Free software (MIT license).

Blocking: Divide and Conquer

Blosc achieves very high speeds by making use of the well-known blocking technique:



Pros and Cons of Blocking

Very fast

Compresses/decompresses at L1 cache speeds.

Lesser compression ratio

The block is the maximum extent in which redundant data can be identified and de-dup'd.

Pros and Cons of Blocking

Very fast

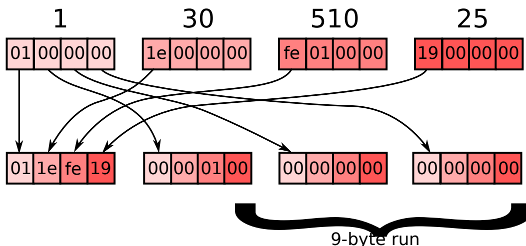
Compresses/decompresses at L1 cache speeds.

Lesser compression ratio

The block is the maximum extent in which redundant data can be identified and de-dup'd.

Shuffling: Improving the Compression Ratio

- Numerical datasets normally can achieve better compression ratios by applying a technique called "*shuffling*" before compressing.
- The shuffling algorithm does not actually compress the data; it rather changes the byte order in the data stream:



Outline

1. Motivation
2. The Data Access Issue
   - A Bit of (Personal) Computing History
   - CPU Starvation and The Hierarchical Memory Model
   - Techniques For Fighting CPU Starvation
3. The Role Of Compression In Data Access
   - Eliminating Data Redundancy
   - Blosc: A BLOcking Shuffler & Compressor
   - An Application of Blosc: the PyTables Database

tables.Expr Calculation Revisited

Now that we have a shiny new compressor library in our tool-set, let us see how it compares with Zlib and LZO for out-of-core computations:

Method	Time (s)	Speed-up	Memory (MB)
numpy.memmap	166	1x	11571
tables.Expr (no compr)	141	1.2x	178
tables.Expr (Zlib)	78	2.1x	180
tables.Expr (LZO)	57	2.9x	180
tables.Expr(Blosc)	41	4.0x	180

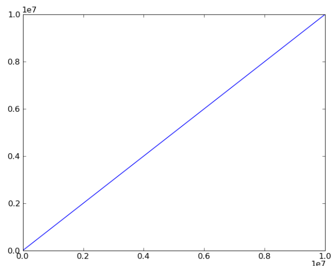
Blosc can make a large difference processing very large datasets!

Blosc and In-Memory Datasets

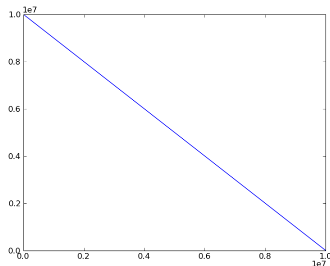
- Several benchmarks have been conducted in order to analyze how Blosc performs in comparison with other compressors when data is *in-memory*.
- The benchmarks consist in reading a couple of datasets from OS filesystem cache, operating upon them and writing the result to the filesystem cache again.
- Datasets analyzed are synthetic (low entropy, so highly compressibles) and real-life (medium/high entropy, difficult to compress well), in both single and double-precision versions.
- Synthetic datasets do represent important corner use cases: sparse matrices, regular grids...

The Synthetic Datasets

These were easy to generate:



$y = x$

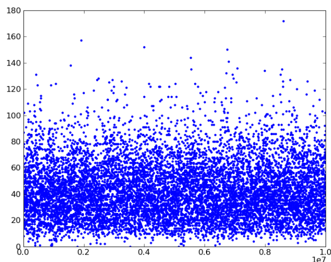


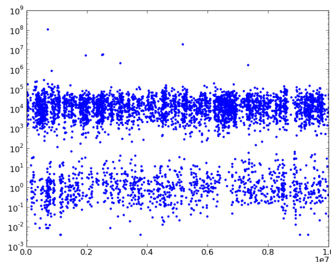
$y' = N - x$

Both datasets are vectors with 10 million elements.

The Real-Life Datasets

These have been taken from drug discovery tests:





(*Source: Toby Mathieson / Cellzome*)

Both datasets are vectors with 10 million elements.

The Expression to Be Computed

- The expression to be computed for the benchmarks is:

  ```
  r = 3*a - 2*b + 1.1
  ```

  where "a" and "b" are the datasets chosen that are preloaded on the filesystem cache , and "r" is the result that will also be written to filesystem cache too.

Compressors Used

The next compression libraries have been used:

- **Zlib:** good compression ratios, normal speed.
- **LZO:** normal compression ratios, but very fast (one of the fastest general-purpose compression libraries I've seen).
- **Blosc:** low compression ratios, extremely fast.

Due to the fact that shuffling improves the compression ratios for the datasets in this benchmark, it has been activated for all the cases.

Compressors Used

The next compression libraries have been used:

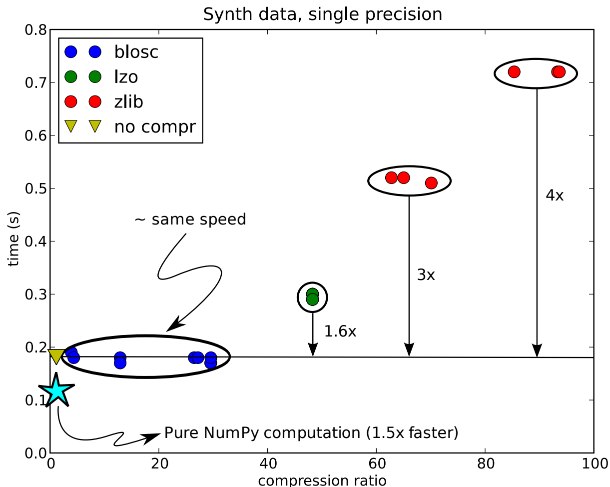
- **Zlib:** good compression ratios, normal speed.
- **LZO:** normal compression ratios, but very fast (one of the fastest general-purpose compression libraries I've seen).
- **Blosc:** low compression ratios, extremely fast.

Due to the fact that shuffling improves the compression ratios for the datasets in this benchmark, it has been activated for all the cases.

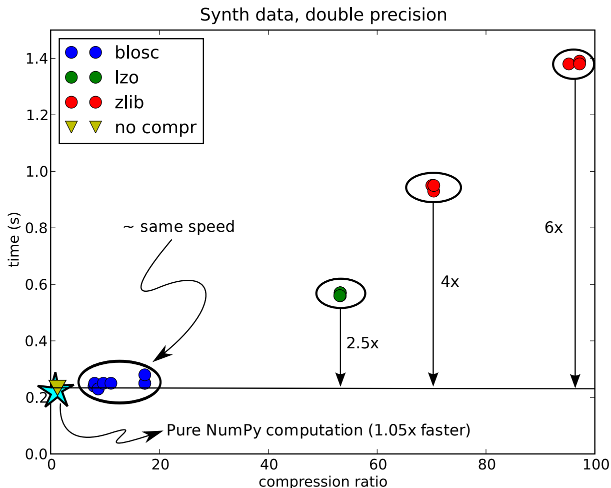
Benchmark Setup

- Intel Core2 machine at 3 GHz and 6 MB L2 cache (dual-core)
- 8 GB of RAM (datasets fit here comfortably)
- RAID-0 of 4 SATA2 disk @ 7200 RPM
- SuSE GNU/Linux 11.1 (x86-64)
- PyTables 2.2 beta1
- HDF5 1.8.2
- NumPy 1.3

Results for Synthetic Data



Results for Synthetic Datasets



Comments for Synthetic Data Benchmarks

- Blosc compresses much less than LZO or Zlib, but also much faster. Both behaviours were expected.
- For this highly compressible data, higher optimization levels don't mean slower overall speed.
- In contrast to the other compressors, Blosc speed is not affected too much by the single- or double-precision data.

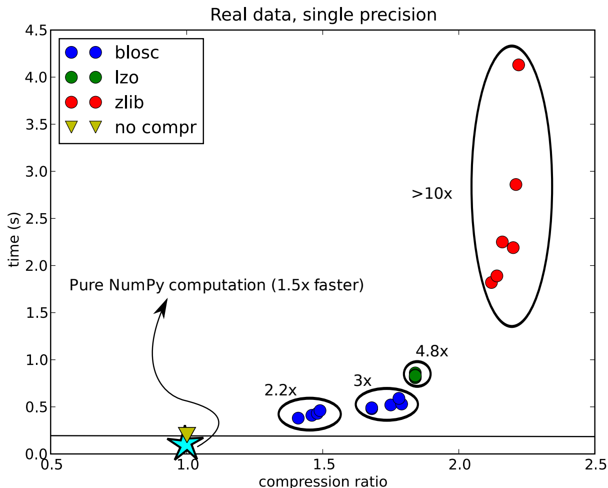
This is the first time (that I'm aware of) that a computation made with compressed data already in RAM (or in the OS filesystem cache, to be precise) matches the speed of the same computation on uncompressed data.

Comments for Synthetic Data Benchmarks

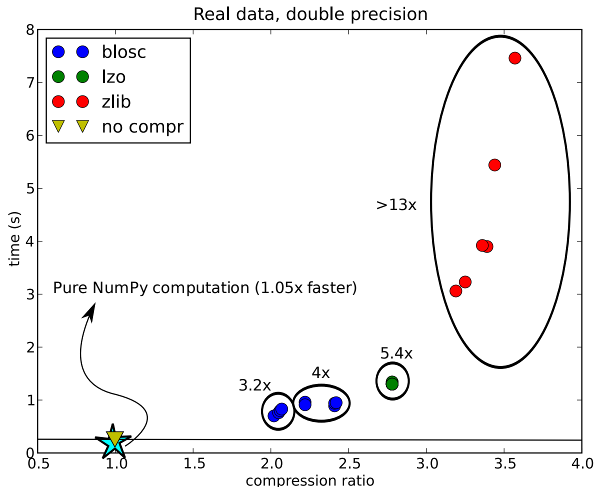
- Blosc compresses much less than LZO or Zlib, but also much faster. Both behaviours were expected.
- For this highly compressible data, higher optimization levels don't mean slower overall speed.
- In contrast to the other compressors, Blosc speed is not affected too much by the single- or double-precision data.

This is the first time (that I'm aware of) that a computation made with compressed data already in RAM (or in the OS filesystem cache, to be precise) matches the speed of the same computation on uncompressed data.

Results for Real Data Scenario



Results for Real Data Scenario



Comments for Real Datasets Benchmarks

- Blosc still compresses less than LZO or Zlib, but the difference is less now. Also, Blosc continues to be **significantly** faster.
- For real data, higher optimization levels can make an important difference, both in terms of time and compression ratio.
- In this case, Blosc speed is noticeably reduced when data is double-precision instead of single.

Blosc cannot match the speed of computations made with uncompressed real data yet. However, it is just a matter of time before it reaches this goal (using either newer processors or more optimized libraries).

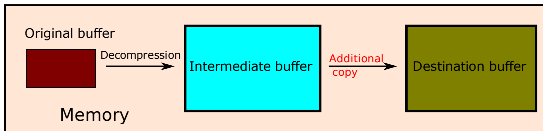
Comments for Real Datasets Benchmarks

- Blosc still compresses less than LZO or Zlib, but the difference is less now. Also, Blosc continues to be **significantly** faster.
- For real data, higher optimization levels can make an important difference, both in terms of time and compression ratio.
- In this case, Blosc speed is noticeably reduced when data is double-precision instead of single.

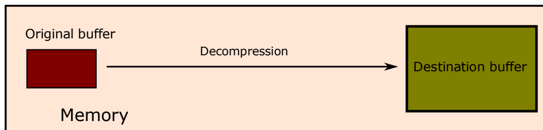
Blosc cannot match the speed of computations made with uncompressed real data yet. However, it is just a matter of time before it reaches this goal (using either newer processors or more optimized libraries).

Side Note: a HDF5 Design Issue

Current behaviour (Extra buffer copy)

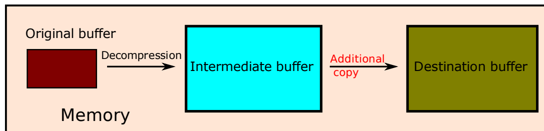


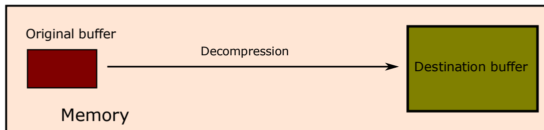
Desired behaviour (No extra copy)



If we could have avoided the extra buffer copy, the achieved speeds in previous benchmarks could have been between a 25% and 50% faster → Blosc can effectively be faster than a regular memcpy (!)

Side Note: a HDF5 Design Issue





If we could have avoided the extra buffer copy, the achieved speeds in previous benchmarks could have been between a 25% and 50% faster → Blosc can effectively be faster than a regular `memcpy` (!)

Summary

- These days, you should understand the hierarchical memory model if you want to get decent performance.
- The hierarchical-aware Blosc compressor offers substantial performance gains over existing compressors (at the cost of compression ratio).
- 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

- Francesc Alted
Blosc: A blocking, shuffling and loss-less compression library
http://blosc.pytables.org

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!).