Highly Efficient Computations In Python

Well Beyond NumPy

Francesc Alted

Freelance Developer and PyTables Creator

July 8th-11th, 2010. Paris - France

Outline

1. The Data Access Issue
   - Why Modern CPUs Are Starving?
   - Techniques For Fighting Data Starvation
   - High Performance Libraries
2. Exercises
3. Answers To Questions In Exercises

Outline

1. The Data Access Issue
   - Why Modern CPUs Are Starving?
   - Techniques For Fighting Data Starvation
   - High Performance Libraries
2. Exercises
3. Answers To Questions In Exercises

Quote Back in 1991

"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, 1991

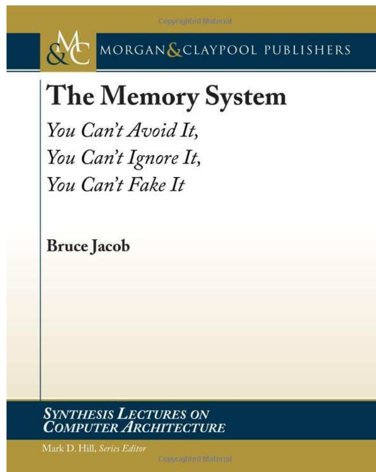
Quote Back in 1996

"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



The CPU Starvation Problem

Know facts (in 2010):

- Memory latency is much slower (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 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: 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?

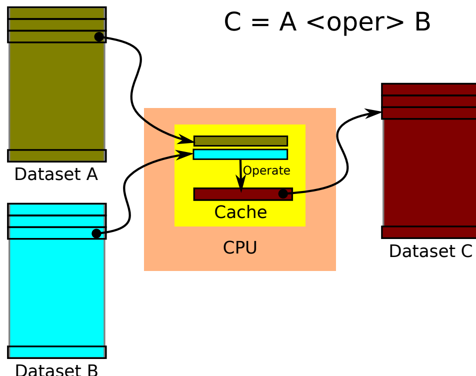
- Caches are closer to the processor (normally in the same die), so both the latency and throughput are improved.
- However: the faster they run the smaller they must be.
- They are effective mainly in a couple of scenarios:
  - Time locality: when the dataset is reused.
  - Spatial locality: when the dataset is accessed sequentially.

Outline

1. The Data Access Issue
   - Why Modern CPUs Are Starving?
   - Techniques For Fighting Data Starvation
   - High Performance Libraries
2. Exercises
3. Answers To Questions In Exercises

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:



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

1. The Data Access Issue
   - Why Modern CPUs Are Starving?
   - Techniques For Fighting Data Starvation
   - High Performance Libraries
2. Exercises
3. Answers To Questions In Exercises

Some High Performance Libraries

MKL (Intel's Math Kernel Library): Uses memory efficient algorithms as well as SIMD and multi-core algorithms $\rightarrow$ 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.

Blosc: A 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 for performing optimal out-of-core computations, as well as fast queries on arbitrarily large tables.

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 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 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
```

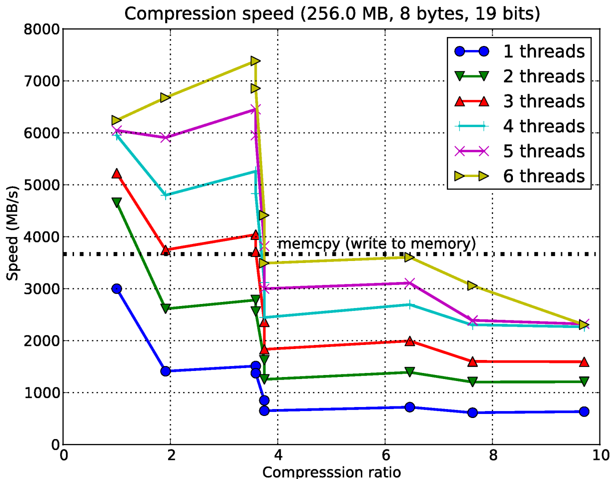
Important

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.

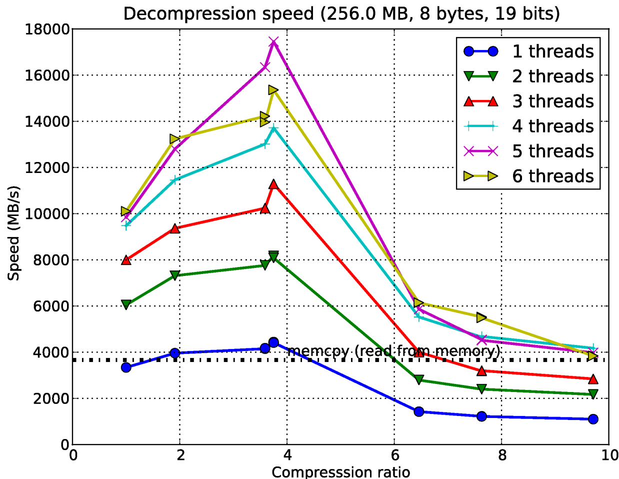
Blosc: A Blocked, Shuffling and Loss-Less Compression Library

- 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 the phase of the moon.
  - Can deal with compressed arrays seamlessly.

Steps To Accelerate Your Code

In order of importance:

- Make use of memory-efficient libraries (most of the current bottlenecks falls into this category).
- Apply the blocking technique and vectorize your code.
- Make use of efficient compressors so as to minimize CPU and memory bandwidth load.
- Use multi-threading (always try this as a last resort).

Time To Some Hands-On

Now that things are so simple, there is so much to do.

– *Morton Feldman*

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!).

Time To Some Hands-On

Go to:

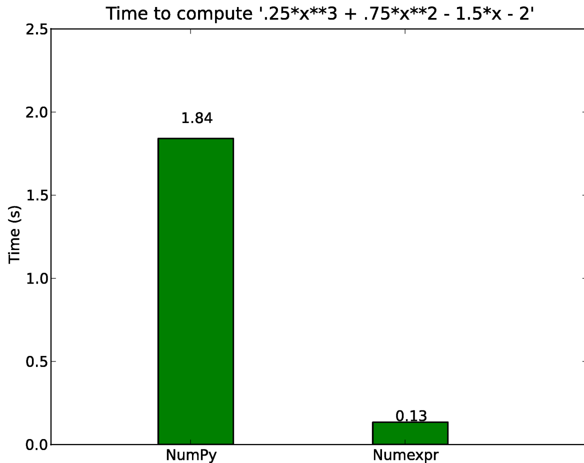
http://www.pytables.org/EuroSciPy2010

and download the `poly1.py` and `poly2.py` scripts.

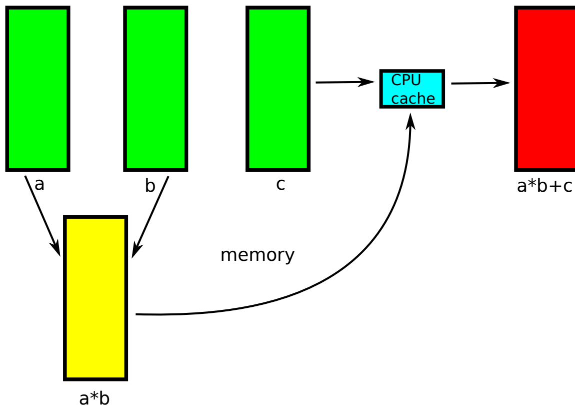
Guidelines

Use the guidelines that have been handout to you (you can download the PDF if you prefer reading electronic documents)

Results For In-Memory Computation (I)

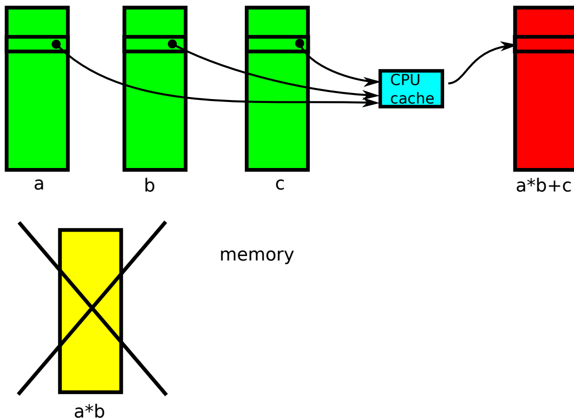


NumPy And Temporaries



Numexpr Avoids (Big) Temporaries

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

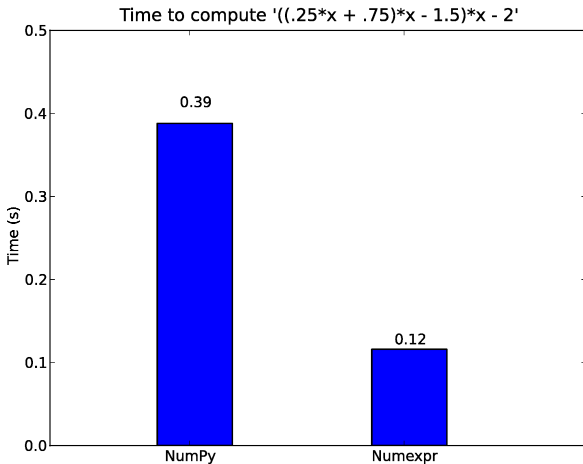


Numexpr Working As A JIT Compiler

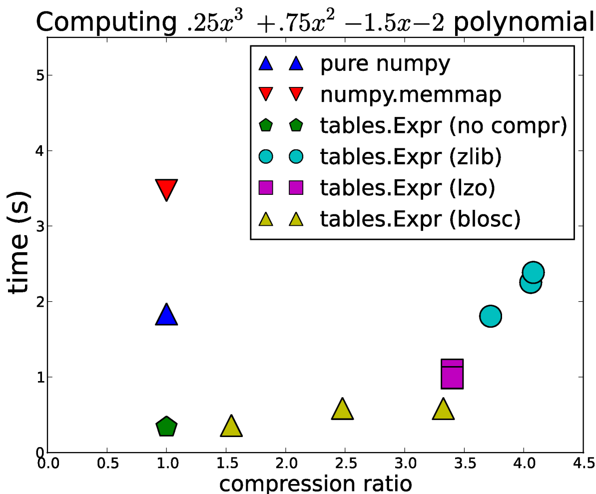
	NumPy	Numexpr
x**3	pow(x,3)	x*x*x
time	23.6 ms	4.32 ms

Numexpr can optimize more scenarios than NumPy can

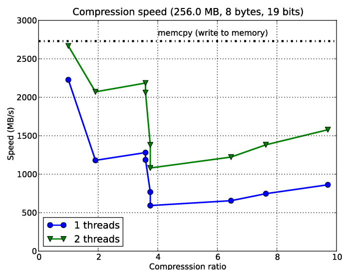
Results For In-Memory Computation (II)

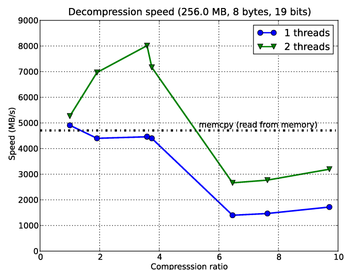


Results For Out-Of-Memory Computation

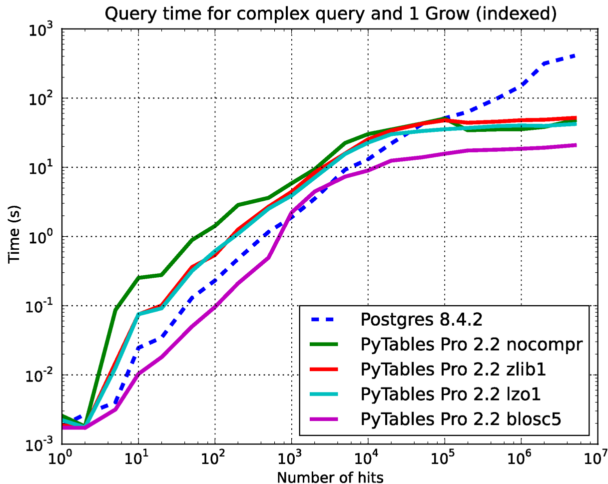


Blosc Can Transmit Data Very Fast

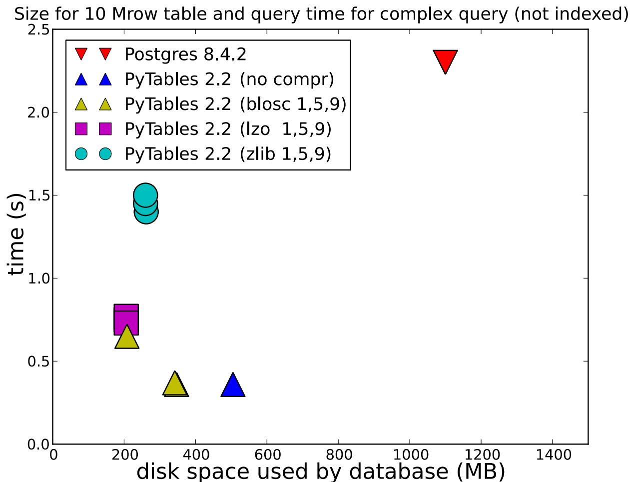




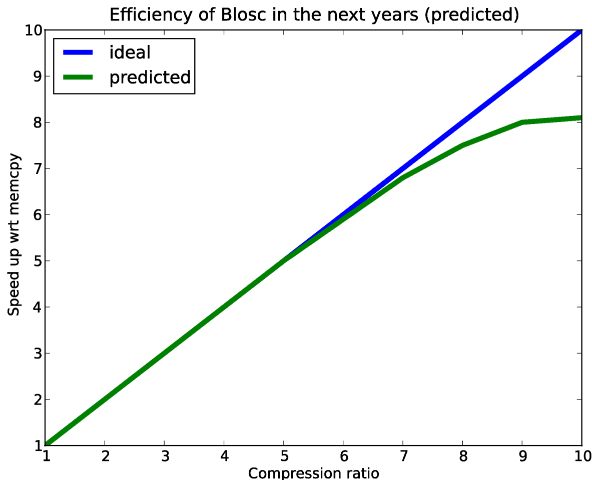
Blosc Accelerates Out-Of-Memory Queries



Blosc To Accelerate In-Memory Queries?



Blosc In Next-Generation CPUs



Summary

- These days, you should take in consideration memory efficiency issues if you want to get decent performance.
- Leverage existing memory-efficient libraries for performing your computations optimally.
- The Blosc compressor offers substantial performance gains over existing compressors, and can even accelerate computations in some situations.
- Try multi-core parallelization only as a last resort (remember: bottlenecks are mostly in memory, not in CPU!)

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

Thank You!

Contact:

faltet@pytables.org