Fast numerical computations with Cython

Dag Sverre Seljebotn
dagss@student.matnat.uio.no

University of Oslo

SciPy2009

Brief introduction to Cython

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly

Cython at a glance

- Cython is used for compiling Python-like code to machine-code
  - supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code (vs. Py2.5, using PyBench)
- In addition:
  - Add types for speedups (hundreds of times)
  - Easily use native libraries (C/C++/Fortran) directly
- How it works: Cython code is turned into C code which uses the CPython API and runtime.
  - Generated C code can be built and run without Cython installed

Coding in Cython is like coding in Python and C at the same time!

Usecase 1: Library wrapping

- Cython is a popular choice for writing Python interface modules for C libraries
- Works very well for writing a higher-level Pythonized wrapper
- For 1:1 wrapping other tools might be better suited, depending on the exact usecase

Usecase 2: Performance-critical code

Python	⟷	C/C++/Fortran
High-level		Lower-level
Slow		Fast
No variables typed		All variables typed

- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python

Usecase 2: Performance-critical code

Python	⟷	C/C++/Fortran
High-level		Lower-level
Slow		Fast
No variables typed		All variables typed

- Common procedure: Where speed is needed, use a compiled language, then wrap the code for use from Python
- Cython: Incremental optimization workflow
  - Optimize, don't re-write
  - Only the pieces you need

Not a usecase: Static type checking

- Cython is (partially) statically typed because it has too, not because it wants to
- You still need to run the program to catch a typo

A bit of history

- April 2002: release of Pyrex 0.1 by Greg Ewing
- Various forks. 28th July 2007: Official Cython launch – merge of lxml's fork (Stefan Behnel) with Sage's fork (Robert Bradshaw, William Stein)
- Summer/autumn 2008: Efficient NumPy support
  - Thanks to Google Summer of Code and Enthought for funding!
- Rapidly growing user base, many from science
- Currently ~ 6 active developers (including two GSoC students)

A numerical example

matmul1 – Python baseline

```
def matmul1(A, B, out=None):
    <...sanity checks, allocate out if needed...>
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
            out[i,j] = s
    return out
```

Compiling in Cython results in ∼1.3x speedup over Python

matmul2 – add types

```
import numpy as np
cimport numpy as np
ctypedef np.float64_t dtype_t
def matmul2(np.ndarray[dtype_t, ndim=2] A,
            np.ndarray[dtype_t, ndim=2] B,
            np.ndarray[dtype_t, ndim=2] out=None):
    cdef Py_ssize_t i, j, k
    cdef dtype_t s
    if A is None or B is None: raise ValueError(<...>)
    <...sanity checks, allocate out if needed...>
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
            out[i,j] = s
    return out
```

Result (in-cache): ∼150x speedup over Python

matmul3 – no wraparound or bounds checking

```
cimport cython
import numpy as np
cimport numpy as np
ctypedef double dtype_t

@cython.boundscheck(False)
@cython.wraparound(False)
def matmul3(np.ndarray[dtype_t, ndim=2] A,
            np.ndarray[dtype_t, ndim=2] B,
            np.ndarray[dtype_t, ndim=2] out=None):
    <...snip...>
```

Result (in-cache): ~620x speedup over Python

Out-of-cache: Array layout matters!

- Accessing arrays in a good order $\Rightarrow$ less jumping around in memory $\Rightarrow$ faster execution in out-of-cache situations.

Out-of-cache: Array layout matters!

- Accessing arrays in a good order $\Rightarrow$ less jumping around in memory $\Rightarrow$ faster execution in out-of-cache situations.
- In matmul, we access the rows of `A` and columns of `B`, so the optimal layout is to have `A` stored with contiguous rows ("C order") and `B` stored with contiguous columns ("Fortran order").

Assuming `X`, `Y` and `out` are C-contiguous (the NumPy default):

	80x80 (50 KB)	600x600 (2.75 MB)
`matmul3(X, Y.T, out)`	1.4 ms	1.0 s
`matmul3(X, Y, out)`	1.4 ms	1.9 s
`matmul3(X.T, Y, out)`	1.4 ms	6.7 s
`matmul3(X.T, Y, out.T)`	1.5 ms	6.7 s

Taking care to copy arrays

```
def matmul(<...>):
    <... sanity checks ...>
    if not A.flags.c_contiguous: A = A.copy('C')
    if not B.flags.f_contiguous: B = B.copy('F')
    orig_out = out
    if out is None or not out.flags.c_contiguous:
        out = out.empty((A.shape[0], B.shape[1]), A.dtype)
    <... do matmul ...>
    if orig_out is not None and orig_out is not out:
        orig_out[:,:] = out
    return out
```

More on array memory layout

NumPy arrays are not necesarrily stored as one contiguous block of memory:

```
A = np.reshape(np.arange(18), (3, 6))
```

A =

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

	A
Start:	0
Shape:	(3, 6)
Strides:	(6, 1)
A[1,2] at:	$0 + 1 \cdot 6 + 2 \cdot 1 = 7$

More on array memory layout

NumPy arrays are not necesarrily stored as one contiguous block of memory:

```
A = np.arange(18); A.shape = (3, 6); B = A[0::2, 5:0:-2]
```

$$\texttt{A} = \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 12 & 13 & 14 & 15 & 16 & 17 \\ \hline \end{array} \quad \Rightarrow \quad \texttt{B} = \begin{array}{|c|c|c|} \hline 5 & 3 & 1 \\ \hline 17 & 15 & 13 \\ \hline \end{array}$$

	`A`	`B`
Start:	0	5
Shape:	(3, 6)	(2, 3)
Strides:	(6, 1)	(12, -2)
Element `[1,2]` at:	$0 + 1 \cdot 6 + 2 \cdot 1 = 8$	$5 + 1 \cdot 12 + 2 \cdot (-2) = 13$

More on array memory layout

NumPy arrays are not necesarrily stored as one contiguous block of memory:

```
A = np.arange(18); A.shape = (3, 6); B = A[0::2, 5:0:-2]
```

$$\mathtt{A} = \begin{array}{|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 6 & 7 & 8 & 9 & 10 & 11 \\ \hline 12 & 13 & 14 & 15 & 16 & 17 \\ \hline \end{array} \Rightarrow \mathtt{B} = \begin{array}{|c|c|c|}\hline 5 & 3 & 1 \\ \hline 17 & 15 & 13 \\ \hline \end{array}$$

	A	B
Start:	0	5
Shape:	(3, 6)	(2, 3)
Strides:	(6, 1)	(12, -2)
Element [1,2] at:	$0 + 1 \cdot 6 + \mathbf{2} \cdot \mathbf{1} = 8$	$5 + 1 \cdot 12 + 2 \cdot (-2) = 13$

Array access method

If one knows compile-time that the array is contiguous, one can save one stride multiplication operation per array access.

```
def matmul4(np.ndarray[dtype_t, ndim=2, mode="c"] A,
            np.ndarray[dtype_t, ndim=2, mode="fortran"] B,
            np.ndarray[dtype_t, ndim=2, mode="c"] out=None)
    <...snip...>
```

This assumes that the arguments have the right layout:

```
>>> matmul4(A, B, out)
Traceback (most recent call last):
    ...
ValueError: ndarray is not Fortran contiguous
```

Solution: Apply the copying strategy shown *before* assigning to typed variables.

Result (in-cache): ~780x speedup over Python

Calling external libraries

- In reality, one would use BLAS for this kind of job.
  - (BLAS is an API for doing linear algebra computations; different implementations available.)

Calling external libraries

- In reality, one would use BLAS for this kind of job.
  - (BLAS is an API for doing linear algebra computations; different implementations available.)
- Let's go half the way and replace the inner products (the inner loop) with a call to BLAS
  - Will use SSE instruction sets if the arrays have the right memory layout, so can expect substantial improvement

Calling external libraries

- In reality, one would use BLAS for this kind of job.
  - (BLAS is an API for doing linear algebra computations; different implementations available.)
- Let's go half the way and replace the inner products (the inner loop) with a call to BLAS
  - Will use SSE instruction sets if the arrays have the right memory layout, so can expect substantial improvement
- Cython code can call BLAS directly without any intermediaries. Declaration needed:

```
cdef extern from "cblas.h":
    double ddot "cblas_ddot"(int N, double *X, int incX,
                             double *Y, int incY)
```

  - Also need to set up the build to link with BLAS.

matmul6 – call BLAS

```
<...snip...>
def matmul6(np.ndarray[dtype_t, ndim=2] A,
            np.ndarray[dtype_t, ndim=2] B,
            np.ndarray[dtype_t, ndim=2] out):
    cdef Py_ssize_t i, j, k
    cdef dtype_t s
    cdef np.ndarray[dtype_t, ndim=1] A_row, B_col
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            A_row = A[i,:]; B_col = B[:,j]
            out[i,j] = ddot(A_row.shape[0],
                            <dtype_t*>A_row.data,
                            A_row.strides[0] // sizeof(dtype_t),
                            <dtype_t*>B_col.data,
                            B_col.strides[0] // sizeof(dtype_t))
```

But: Much slower (for small n), due to extra Python object construction

matmul6 – call BLAS w/ pointer arithmetic

Some low-level C pointer arithmetic saves the day:

```
<...snip...>
def matmul6(np.ndarray[dtype_t, ndim=2] A,
            np.ndarray[dtype_t, ndim=2] B,
            np.ndarray[dtype_t, ndim=2] out):
    cdef Py_ssize_t i, j, k
    cdef dtype_t s
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            out[i,j] = ddot(A.shape[1],
                            <dtype_t*>(A.data + i*A.strides[0]),
                            A.strides[1] // sizeof(dtype_t),
                            <dtype_t*>(B.data + j*B.strides[1]),
                            B.strides[0] // sizeof(dtype_t))
```

Result (in-cache): ~1350x speedup over Python

Benchmarks

	80x80 (50 KB)		600x600 (2.75 MB)	
	MFLOPS	(speedup)	MFLOPS	(speedup)
Optimal layout				
Python	1.14		1.21	
Cython	1.53	(1.35x)	1.54	(1.27x)
Added types	167	(147x)	164	(136x)
`boundscheck`/`wraparound`	711	(626x)	450	(372x)
`mode="c"`/`mode="fortran"`	890	(784x)	474	(391x)
BLAS `ddot` (ATLAS)	1530	(1348x)	574	(474x)
gfortran $A^T B$	914	(805x)	497	(410x)
Intel Fortran $A^T B$	2440	(2148x)	608	(503x)
Worst-case layout				
Python	1.12		1.2	
`boundscheck`/`wraparound`	644	(575x)	63.7	(53x)
BLAS `ddot` (ATLAS)	737	(658x)	63.6	(53x)
gfortran AB^T	868	(775x)	64.9	(54x)
Intel Fortran AB^T	875	(781x)	65.0	(54x)

AMD Athlon 64 X2 3800+, 512 KB cache, SSE2.

Parallel computations

Shared memory model

- OpenMP not available with Cython

Shared memory model

- OpenMP not available with Cython
- Python threads can be used
  - Much inter-thread communication? Possible to use native OS thread API in combination for speedup (with some care)

Shared memory model

- OpenMP not available with Cython
- Python threads can be used
  - Much inter-thread communication? Possible to use native OS thread API in combination for speedup (with some care)
- Problem: By default, thread switching does not occur within Cython code
  - Due to Python's Global Interpreter Lock (GIL)
  - $\Rightarrow$ `matmul`s below execute in serial
  - Actually worse than pure Python...

```
from previous_part import matmul2
from threading import Thread
args = [(A1, B1, out1), (A2, B2, out2), ...]
threads = [Thread(target=matmul2, args=x) for x in args]
for t in threads: t.start()
for t in threads: t.join()
```

Releasing the GIL solves the issue

```
@cython.boundscheck(False)
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out=None):
    cdef Py_ssize_t i, j, k
    cdef dtype_t s
    <...sanity checks, allocate out if needed...>
    with nogil:
        for i from 0 <= i < A.shape[0]:
            for j from 0 <= j < B.shape[1]:
                s = 0
                for k from 0 <= k < A.shape[1]:
                    s += A[i, k] * B[k, j]
            out[i,j] = s
    return out
```

- Cannot raise exceptions in nogil section ⇒ boundscheck(False)
- Change of for-loop syntax not necesarry in next Cython release

Message passing / grid computing

- No GIL issue
- Popular approach: MPI and mpi4py
  - mpi4py (Lisandro Dalcin et al.) is an example of a library written in Cython

Mixing mpi4py and C MPI

- If you have lots and lots of small messages: Possible to mix with the C MPI API
- mpi4py ships Cython compile-time definitions

```
from mpi4py import MPI
from mpi4py cimport MPI
from mpi4py cimport mpi_c

cdef MPI.Comm comm = MPI.COMM_WORLD

if comm.Get_rank() == 0:
    comm.send({'a': any_python_object, 'b': other}, to=1)
    for <...a lot of small packages...>:
        mpi_c.MPI_Send(..., comm.ob_mpi)
elif ...
```

Ongoing and future work

In progress

Two Google Summer of Code projects this summer:

- Better Fortran support, “memoryviews” (Kurt Smith; next talk!)
- Better C++ support (Danilo Freitas)

In progress

Two Google Summer of Code projects this summer:

- Better Fortran support, “memoryviews” (Kurt Smith; next talk!)
- Better C++ support (Danilo Freitas)

Also in progress:

- Closures (inner functions, lambda, generators)!
- Profiling support
- Pure Python mode(s)

New memoryview type (details may change)

- In Python 2.6+ any object can export a multidimensional underlying buffer (PEP 3118)
- A lower-level, more explicit way of treating array data
  - Start to think in terms of any buffer (including C arrays), not only NumPy arrays

```
def matmul(double[:,:] A,
           double[:,:] B,
           double[:,:] out = None):
    <... same as before ...>
    # can no longer do e.g. A.mean() (if we wanted)

matmul(PIL_image, cython.carray_from(c_ptr, (n,m)))
```

New memoryview type (details may change)

- In Python 2.6+ any object can export a multidimensional underlying buffer (PEP 3118)
- A lower-level, more explicit way of treating array data
  - Start to think in terms of any buffer (including C arrays), not only NumPy arrays
  - Easy assumptions on contiguousness w/automatic copying

```
def matmul(in double[:,::contig] A,
           in double[::contig,:] B,
           out double[:,::contig] out = None):
    <... same as before ...>
```

```
matmul(A[::3,:], B.T[:,::3]) # still contiguous inner axis!
```

New memoryview type (details may change)

- In Python 2.6+ any object can export a multidimensional underlying buffer (PEP 3118)
- A lower-level, more explicit way of treating array data
  - Start to think in terms of any buffer (including C arrays), not only NumPy arrays
  - Easy assumptions on contiguousness w/automatic copying
  - Enables very fast slicing (no room in syntax otherwise...)

```
def matmul(in double[::contig,:] A,
           in double[:,::contig] B,
           out double[::contig,:] out = None):
    <...>
    cdef double[::contig] A_row, B_col
    for i in range(A.shape[0]): for j in range(B.shape[1]):
        A_row = A[i,:]; B_col = B[:,j]
        out[i,j] = ddot(a.shape[0], ...)
```

Future: Which way?

- This also opens up the syntax for (possibly) native Cython SIMD operations:

```
cdef extern from "math.h":
    double sqrt(double)
def distance(double[:] x, double[:] y):
    return sqrt(x**2 + y**2)
```

Future: Which way?

- This also opens up the syntax for (possibly) native Cython SIMD operations:

```
cdef extern from "math.h":
    double sqrt(double)
def distance(double[:] x, double[:] y):
    return sqrt(x**2 + y**2)
```

- Benchmarks indicate a 4x speedup over NumPy (out-of-cache) for this particular example (lower bound, using a trivial C loop)
    - Basic C loop implementation *not difficult*
    - Then one can add plugin backends... Blitz++, TooN, PyCUDA, ...

Future: Which way?

- This also opens up the syntax for (possibly) native Cython SIMD operations:

```
cdef extern from "math.h":
    double sqrt(double)
def distance(double[:] x, double[:] y):
    return sqrt(x**2 + y**2)
```

- Benchmarks indicate a 4x speedup over NumPy (out-of-cache) for this particular example (lower bound, using a trivial C loop)
  - Basic C loop implementation *not difficult*
  - Then one can add plugin backends... Blitz++, TooN, PyCUDA, ...
- Pro: Gets Cython w/NumPy a good deal closer to the speed/convenience of Fortran (without some of the issues...)
- Con: Should focus any efforts on more flexible run-time approaches?

Myself I am not sure yet...

Cython & Fwrap BoF at 6:30

Keck (opposite to Powell Booth, room 111)

Everybody welcome!