Fast numerical computations with Cython

Dag Sverre Seljebotn (dagss@student.matnat.uio.no) – *University of Oslo*[a b c], Norway

[a]Institute of Theoretical Astrophysics, University of Oslo, P.O. Box 1029 Blindern, N-0315 Oslo, Norway
[b]Department of Mathematics, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway
[c]Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway

Cython has recently gained popularity as a tool for conveniently performing numerical computations in the Python environment, as well as mixing efficient calls to natively compiled libraries with Python code. We discuss Cython's features for fast NumPy array access in detail through examples and benchmarks. Using Cython to call natively compiled scientific libraries as well as using Cython in parallel computations is also given consideration. We conclude with a note on possible directions for future Cython development.

Introduction

Python has in many fields become a popular choice for scientific computation and visualization. Being designed as a general purpose scripting language without a specific target audience in mind, it tends to scale well as simple experiments grow to complex applications. From a numerical perspective, Python and associated libraries can be regarded mainly as a convenient shell around computational cores written in natively compiled languages, such as C, C++ and Fortran. For instance, the Python-specific SciPy [SciPy] library contains over 200 000 lines of C++, 60 000 lines of C, and 75 000 lines of Fortran, compared to about 70 000 lines of Python code.

There are several good reasons for such a workflow. First, if the underlying compiled library is usable in its own right, and also has end-users writing code in MATLAB, C++ or Fortran, it may make little sense to tie it too strongly to the Python environment. In such cases, writing the computational cores in a compiled language and using a Python wrapper to direct the computations can be the ideal workflow. Secondly, as we will see, the Python interpreter is too slow to be usable for writing low-level numerical loops. This is particularly a problem for computations which can not be expressed as operations on entire arrays.

Cython is a programming language based on Python, with additional syntax for optional static type declarations. The Cython compiler is able to translate Cython code into C code making use of the CPython C API [CPyAPI], which can in turn be compiled into a module loadable into any CPython session. The end-result can perhaps be described as a language which allows one to use Python and C interchangeably in the same code. This has two important applications. First, it is useful for creating Python wrappers around natively compiled code, in particular in situations where one does not want a 1:1 mapping between the library API and the Python API, but rather a higher-level Pythonic wrapper. Secondly, it allows incrementally speeding up Python code. One can start out with a simple Python prototype, then proceed to incrementally add type information and C-level optimization strategies in the few locations that really matter. While being a superset of Python is a goal for Cython, there is currently a few incompatibilities and unsupported constructs. The most important of these is inner functions and generators (closure support).

In this paper we will discuss Cython from a numerical computation perspective. Code is provided for illustration purposes and the syntax is not explained in full, for a detailed introduction to Cython we refer to [Tutorial] and [Docs]. [Wilbers] compares Cython with similar tools ([f2py], [Weave], [Instant] and [Psyco]). The comparison is for speeding up a particular numerical loop, and both speed and usability is discussed. Cython here achieves a running time 1.6 times that of the Fortran implementation. We note that had the arrays been declared as contiguous at compile-time, this would have been reduced to 1.3 times the time of Fortran. [Ramach] is a similar set of benchmarks, which compare Pyrex and other tools with a pure Python/NumPy implementation. Cython is based on [Pyrex] and the same results should apply, the main difference being that Cython has friendlier syntax for accessing NumPy arrays efficiently.

Fast array access

Fast array access, added to the Cython language by D. S. Seljebotn and R. W. Bradshaw in 2008, was an important improvement in convenience for numerical users. The work is based on PEP 3118, which defines a C API for direct access to the array data of Python objects acting as array data containers[1]. Cython is able to treat most of the NumPy array data types as corresponding native C types. Since Cython 0.11.2, complex floating point types are supported, either through the C99 complex types or through Cython's own implementation. Record arrays are mapped to arrays of C structs for efficient access. Some data types are not supported, such as string/unicode arrays, arrays with non-native endianness and boolean arrays. The latter can however be treated as 8-bit integer arrays in Cython. [Tutorial] contains further details.

[1]PEP 3118 is only available on Python 2.6 and greater, therefore a backwards-compatibility mechanism is also provided to

To discuss this feature we will start with the example of naive matrix multiplication. Beginning with a pure Python implementation, we will incrementally add optimizations. The benchmarks should help Cython users decide how far one wants to go in other cases. For $C = AB$ the computation is

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where n is the number of columns in A and rows in B. A basic implementation in pure Python looks like this:

```
def matmul(A, B, out):
    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
```

For clarity of exposition, this skips the details of sanity checking the arguments. In a real setting one should probably also automatically allocate `out` if not provided by the caller.

Simply compiling this in Cython results in a about 1.15x speedup over Python. This minor speedup is due to the compiled C code being faster than Python's byte code interpreter. The generated C code still uses the Python C API, so that e.g. the array lookup `A[i, k]` translates into C code very similar to:

```
tmp = PyTuple_New(2);
if (!tmp) { err_lineno = 21; goto error; }
Py_INCREF(i);
PyTuple_SET_ITEM(tmp, 0, i);
Py_INCREF(k);
PyTuple_SET_ITEM(tmp, 1, k);
A_ik = PyObject_GetItem(A, tmp);
if (!A_ik) { err_lineno = 21; goto error; }
Py_DECREF(tmp);
```

The result is a pointer to a Python object, which is further processed with `PyNumber_Multiply` and so on. To get any real speedup, types must be added:

```
import numpy as np
cimport numpy as np
ctypedef np.float64_t dtype_t
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
  if A is None or B is None:
      raise ValueError("Input matrix cannot be None")
  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
```

In our benchmarks this results in a speedup of between 180-190 times over pure Python. The exact factor varies depending on the size of the matrices. If they are not small enough to be kept in the CPU cache, the data must be transported repeatedly over the memory bus. This is close to equally expensive for Python and Cython and thus tends to slightly diminish any other effects. Table 1 has the complete benchmarks, with one in-cache benchmark and one out-of-cache benchmark in every case.

emulate the protocol on older Python versions. This mechanism is also used in the case of NumPy arrays, which do not yet support PEP 3118, even on Python 2.6.

Note however that the speedup does not come without some costs. First, the routine is now only usable for 64-bit floating point. Arrays containing any other data type will result in a `ValueError` being raised. Second, it is necessary to ensure that typed variables containing Python objects are not `None`. Failing to do so can result in a crash or data corruption if `None` is passed to the routine.

The generated C source for the array lookup `A[i, k]` now looks like this:

```
tmp_i = i; tmp_k = k;
if (tmp_i < 0) tmp_i += A_shape_0;
if (tmp_i < 0 || tmp_i >= A_shape_1) {
  PyErr_Format(<...>);
  err_lineno = 33; goto error;
}
if (tmp_k < 0) tmp_k += A_shape_1;
if (tmp_k < 0 || tmp_k >= A_shape_1) {
  PyErr_Format(<...>);
  err_lineno = 33; goto error;
}
A_ik = *(dtype_t*)(A_data +
  tmp_i * A_stride_0 + tmp_k * A_stride_1);
```

This is a lot faster because there are no API calls in a normal situation, and access of the data happens directly to the underlying memory location. The initial conditional tests are there for two reasons. First, an if-test is needed to support negative indices. With the usual Python semantics, `A[-1, -1]` should refer to the lower-right corner of the matrix. Second, it is necessary to raise an exception if an index is out of bounds.

Such if-tests can bring a large speed penalty, especially in the middle of the computational loop. It is therefore possible to instruct Cython to turn off these features through compiler directives. The following code disables support for negative indices (`wraparound`) and bounds checking:

```
cimport cython
@cython.boundscheck(False)
@cython.wraparound(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):
    <...>
```

This removes all the if-tests from the generated code. The resulting benchmarks indicate around 800 times speedup at this point in the in-cache situation, 700 times out-of-cache. Only disabling one of either `wraparound` or `boundscheck` will not have a significant impact because there will still be if-tests left.

One trade-off is that should one access the arrays out of bounds, one will have data corruption or a program crash. The normal procedure is to leave bounds checking on until one is completely sure that the code is correct, then turn it off. In this case we are not using negative indices, and it is an easy decision to turn them off. Even if we were, it would be faster to manually add code to calculate the corresponding positive index. `wraparound` is mainly enabled by default to reduce the number of surprises for casual Cython users, and one should rarely leave it on.

In addition to per-function level like here, compiler directives can also be specified globally for the source file or for individual code blocks. See [directives] for further information.

	80x80	1500x1500
	Units:	MFLOPS
Optimal layout		
Python	0.94	0.98
Cython	1.08	1.12
Added types	179	177
`boundscheck/wraparound`	770	692
`mode="c"/mode="fortran"`	981	787
BLAS `ddot` (ATLAS)	1282	911
Intel C	2560	1022
gfortran A^TB	1113	854
Intel Fortran A^TB	2833	1023
NumPy `dot`	3656	4757
Worst-case layout		
Python	0.94	0.97
`boundscheck/wraparound`	847	175
BLAS `ddot` (ATLAS)	910	183
gfortran AB^T	861	94
Intel Fortran AB^T	731	94

***Table 1**: Matrix multiplication benchmarks on an Intel Xeon 3.2 GHz, 2 MB cache, SSE2. The smaller data set fits in cache, while the larger does not. Keep in mind that different implementations have different constant-time overhead, which e.g. explains that NumPy `dot` does better for larger dataset.*

Caring about memory layout

Both C/C++ and Fortran assume that arrays are stored as one contiguous chunk in memory. NumPy arrays depart from this tradition and allows for arbitrarily strided arrays. Consider the example of `B = A[::-2,:]`. That is, let B be the array consisting of every other row in A, in reverse order. In many environments, the need for representing B contiguously in memory mandates that a copy is made in such situations. NumPy supports a wider range of array memory layouts and can in this situation construct B as a new view to the same data that A refers to. The benefit of the NumPy approach is that it is more flexible, and allows avoiding copying of data. This is especially important if one has huge data sets where the main memory might only be able to hold one copy at the time. With choice does however come responsibility. In order to gain top performance with numerical computations it is in general crucial to pay attention to memory layout.

In the example of matrix multiplication, the first matrix is accessed row-by-row and the second column-by-column. The first matrix should thus be stored with contiguous rows, "C contiguous", while the second should be stored with contiguous columns, "Fortran contiguous". This will keep the distance between subsequent memory accesses as small as possible. In an out-of-cache situation, our fastest matmul routine so far does around 700 times better than pure Python when presented with matrices with optimal layout, but only around 180 times better with the worst-case layout. See table 1.

The second issue concerning memory layout is that a price is paid for the generality of the NumPy approach: In order to address arbitrarily strided arrays, an extra integer multiplication operation must be done per access. In the `matmul` implementation above, `A[i,k]` was translated to the following C code:

```
A_ik = *(dtype_t*)(A_data + i * A_stride_0
                          + j * A_stride_1);
```

By telling Cython at compile-time that the arrays are contiguous, it is possible to drop the innermost stride multiplication. This is done by using the "mode" argument to the array type:

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

The in-cache benchmark now indicate a 780x speedup over pure Python. The out-of-cache improvement is smaller but still noticeable. Note that no restrictions is put on the `out` argument. Doing so did not lead to any significant speedup as `out` is not accessed in the inner loop.

The catch is, of course, that the routine will now reject arrays which does not satisfy the requirements. This happens by raising a ValueError. One can make sure that arrays are allocated using the right layout by passing the "order" argument to most NumPy array constructor functions, as well as the copy() method of NumPy arrays. Furthermore, if an array A is C-contiguous, then the transpose, A.T, will be a Fortran-contiguous view of the same data.

The number of memory accesses of multiplying two $n \times n$ matrices scale as $O(n^3)$, while copying the matrices scale as $O(n^2)$. One would therefore expect, given that enough memory is available, that making a temporary copy pays off once n passes a certain threshold. In this case benchmarks indicate that the threshold is indeed very low (in the range of $n = 10$) and one would typically copy in all situations. The NumPy functions `ascontiguousarray` and `asfortranarray` are helpful in such situations.

Calling an external library

The real world usecase for Cython is to speed up custom numerical loops for which there are no prior implementations available. For a simple example like matrix multiplication, going with existing implementations is always better. For instance, NumPy's `dot` function is about 6 times faster than our fastest Cython implementation, since it uses smarter algorithms. Under the hood, `dot` makes a call to the `dgemm` function in the Basic Linear Algebra Software API ([BLAS], [ATLAS])[2]. One advantage of Cython is how easy it is to call native code. Indeed, for many, this is the entire point of using Cython. We will demonstrate these features by calling BLAS for the inner products only, rather than for the whole matrix multiplication. This allows us to stick with the naive matrix multiplication algorithm, and also demonstrates how to mix Cython code and the use of external libraries. The BLAS API must first be declared to Cython:

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

The need to re-declare functions which are already declared in C is unfortunate and an area of possible improvement for Cython. Only the declarations that are actually used needs to be declared. Note also the use of C pointers to represent arrays. BLAS also accepts strided arrays and expects the strides passed in the `incX` and `incY` arguments. Other C APIs will often require a contiguous array where the stride is fixed to one array element; in the previous section it was discussed how one can ensure that arrays are contiguous. The matrix multiplication can now be performed like this:

```
ctypedef np.float64_t dtype_t
def matmul(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
  cdef np.ndarray[dtype_t, ndim=1] A_row, B_col
  for i in range(A.shape[0]):
      A_row = A[i,:]
      for j in range(B.shape[1]):
          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))
```

This demonstrates how NumPy array data can be passed to C code. Note that NumPy strides are in number of bytes, while BLAS expects them in number of elements. Also, because the array variables are typed, the "data" attributes are C pointers rather than Python buffer objects.

Unfortunately, this results in a slowdown for moderate n. This is due to the slice operations. Operations like `A_row = A[i,:]` is a Python operation and results in Python call overhead and the construction of several new objects. Cython is unlikely to ever optimize slicing of np.ndarray variables because it should remain possible to use subclasses of ndarray with a different slicing behaviour[3]. The new memory view type, discussed below, represents a future solution to this problem. Another solution is to do the slice calculations manually and use C pointer arithmetic:

```
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))
```

This version leads to over 1300 times speedup over pure Python in the optimal, in-cache situation. This is due to BLAS using the SSE2 instruction set, which enables doing two double-precision floating point multiplications in one CPU instruction. When non-contiguous arrays are used the performance drops to 970 times that of pure Python (in-cache) as SSE2 can no longer be used.

For more advanced array data passing, Cython makes it easy to make use of NumPy's C API. Consider for instance a custom C library which returns a pointer to some statically allocated data, which one would like to view as a NumPy array. Using Cython and the NumPy C API this is easily achieved:

```
cimport numpy as np
cdef extern from "mylib.h":
    cdef int get_my_data(double** out_data,
                         int* out_size)
def my_data_as_ndarray():
    cdef np.npy_intp* shape = [0]
    cdef int arr_length
    cdef double* arr_ptr
    if get_my_data(&arr_ptr, &arr_length) != 0:
        raise RuntimeError("get_my_data failed")
    shape[0] = arr_length
    return np.PyArray_SimpleNewFromData(1, shape,
      np.NPY_DOUBLE, <void*>arr_ptr)
```

SSE and vectorizing C compilers

What about using SSE directly in a Cython program? One possibility is to use the SSE API of the C compiler. The details varies according to the C compiler used, but most C compilers offers a set of special functions which corresponds directly to SSE instructions. These can be used as any other C function, also from Cython code. In general, such code tends to be somewhat more complicated, as the first element in every loop must be treated as a special case (in case the elements involved are not aligned on 128-bit boundaries in memory, as required by SSE). We have not included code or benchmarks for this approach.

[2]BLAS is an API with many implementations; the benchmarks in this paper is based on using the open-source ATLAS implementation, custom-compiled on the host by Sage [Sage].

[3]This principle has of course already been violated, as one could change the behaviour of the element indexing in a subclass as well. The policy is however not to go further in this direction. Note also that only indexing with typed integer variables is optimized; A[i, some_untyped_var] is not optimized as the latter index could e.g. point to a Python slice object.

Another popular approach to SSE is to use a "vectorizing" C compiler. For instance both Intel's C compiler, [ICC], and the GNU C compiler, [GCC], can recognize certain loops as being fit for SSE optimization (and other related optimizations). Note that this is a non-trivial task as multiple loop iterations are combined, and the loop must typically be studied as a whole. Unfortunately, neither ICC v. 11 nor GCC v. 4.3.3 managed to vectorize the kind of code Cython outputs by default. After some manual code massaging we managed to have ICC compile a vectorized version which is included in the benchmarks. We did not manage to get GCC to vectorize the kind of sum-reduce loop used above.

It appears that Cython has some way to go to be able to benefit from vectorizing C compilers. Improving Cython so that the generated C code is more easily vectorizable should be possible, but has not been attempted thus far. Another related area of possible improvement is to support generating code containing the C99 restrict modifier, which can be used to provide guarantees that arrays do not overlap. GCC (but not ICC) needs this to be present to be able to perform vectorization.

Linear time: Comparisons with NumPy

We turn to some examples with linear running time. In all cases the computation can easily be expressed in terms of operations on whole arrays, allowing comparison with NumPy.

First, we consider finding elements with a given value in a one-dimensional array. This operation can be performed in NumPy as:

```
haystack = get_array_data()
result = np.nonzero(haystack == 20)
```

This results in a array of indices, listing every element equal to 20. If the goal is simply to extract the first such element, one can instead use a very simple loop in Cython. This avoids constructing a temporary array and the result array. A simple Cython loop then performed about five times faster than the NumPy code in our benchmarks. The point here is merely that Cython allows easily writing code specifically tailored for the situation at hand, which sometimes can bring speed benefits.

Another example is that of operating on a set of array elements matching some filter. For instance, consider transforming all 2D points within a given distance from zero:

```
Point_dtype = np.dtype([('x', np.float64),
                        ('y', np.float64)])
points = load_point_data(filename, Point_dtype)
radius = 1.2
tmp = points['x']**2
tmp += points['y']**2
pointset = tmp < radius**2
points['x'][pointset] *= 0.5
points['y'][pointset] *= 0.3
```

This code uses a number of temporary arrays to perform the calculation. In an in-cache situation, the overhead of constructing the temporary Python arrays becomes noticeable. In an out-of-cache situation, the data has to be transported many times over the memory bus. The situation is worsened by using a record array (as more data is transported over the bus in total, and less cache becomes available). Using separate arrays for `x` and `y` results in a small speedup; both array layouts are included in the benchmarks.

A Cython loop is able to do the operation in a single pass, so that the data is only transported once:

```
cdef packed struct Point:
    np.float64_t x, y

def transform_within_circle(np.ndarray[Point] points,
                            np.float64_t radius):
  cdef Py_ssize_t i
  cdef Point p
  cdef np.float64_t radius_sq = radius**2
  for i in range(points.shape[0]):
      p = points[i]
      if p.x**2 + p.y**2 < radius_sq:
          p.x *= 0.5
          p.y *= 0.3
          points[i] = p
```

This is 10 times faster than the NumPy code for a large data set, due to the heavily reduced memory bus traffic. NumPy also uses twice as much memory due to the temporaries. If the data set is several gigabytes, then the additional memory used by NumPy could mean the difference between swapping and not swapping to disk. For Cython, operating on separate `x` and `y` arrays is slightly slower. See table 2.

Finally, it would have been possible to separate the filter and the transform by passing a callback to be called in each iteration in the loop. By making use of Cython extension type classes, which have faster method dispatches than Python classes, the penalty of such an approach is only around 20-25%. [Tutorial] demonstrates such callbacks.

	Million elements processed per second	
	2×10^7 element test set	
	Records	Seperate
Python loop	0.028	0.069
NumPy	9.5	10
Cython plain	95	79
Cython optimized	110	100
Cython w/callback	79	73

***Table 2**: Benchmarks for operating on points within a circle. The optimized Cython version and the callback version both has `boundscheck`/`wraparound` turned off and `mode='c'` specified. All benchmarks on an Intel Xeon 3.2 GHz, 2 MB cache. All points were within the circle in the test data set.*

Parallel computation

When discussing parallel computation there is an important distinction between shared-memory models and message passing models. We start with discussing the shared memory case. A common approach in parallel numerical code is to use OpenMP. While not difficult to support in principle, OpenMP is currently not available in Cython. Instead, Python threads must be used. This comes with some problems, but they can be worked around.

Threads is a problem with CPython because every operation involving a Python object must be done while holding the Global Interpreter Lock (GIL). The result is that pure Python scripts are typically unable to utilize more than one CPU core, even if many threads are used. It should be noted that for many computational scripts this does not matter. If the bulk of the computation happens in wrapped, native code (like in the case of NumPy or SciPy) then the GIL is typically released during the computation. For Cython the situation is worse. Once inside Cython code, the GIL is by default held until one returns to the Python caller. The effect is that threads doesn't switch at all. Whereas a pure Python script will tend to switch between threads on a single CPU core, a Cython program will by default tend to not switch threads at all.

The solution is to use Cython language constructs to manually release the GIL. One can then achieve proper multi-threading on many cores. The catch is that no Python operations are allowed when the GIL is released; for instance, all variables used must be typed with a C type. Optimized NumPy array lookups are allowed. The Cython compiler will help enforce these rules. Example:

```
@cython.boundscheck(False)
def find_first(np.ndarray[np.int64_t] haystack,
               np.int64_t needle):
    cdef Py_ssize_t i, ret = -1
    with nogil:
        for i from 0 <= i < haystack.shape[0]:
            if haystack[i] == needle:
                ret = i; break
    return ret
```

Without `nogil`, invocations from separate threads would be serialized. Returning the result is a Python operation, so that has to be put outside of the `nogil` block . Furthermore, `boundscheck` must be turned off as raising an `IndexError` would require the GIL[4]. [Docs] contains further information on the various primitives for managing the GIL (search for "nogil").

The message passing case is much simpler. Several Python interpreters are launched, each in its own process, so that the GIL is not an issue. A popular approach is mpi4py [mpi4py], together with an MPI implementation (such as OpenMPI [OpenMPI]). mpi4py very conveniently allows passing full Python objects between computational nodes through Python pickling[5]. It is also possible to efficiently pass NumPy arrays. mpi4py is itself written in Cython, and ships with the Cython compile-time definitions necessary to communicate directly with the underlying MPI C API. It is thus possible to use both interchangeably:

```
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:
    # High-level send of Python object
    comm.send({'a': any_python_object, 'b': other},
              to=1)
    for <...a lot of small C-typed messages...>:
        # Fast, low-level send of typed C data
        mpi_c.MPI_Send(<...>, comm.ob_mpi)
elif ...
```

This is useful e.g. in situations where one wants to pass typed Cython variables and does not want to bother with conversion back and forth to Python objects. This also avoids the overhead of the Python call to mpi4py (although in practice there is likely to be other, larger bottlenecks). While contrived, this example should demonstrate some of the flexibility of Cython with regards to native libraries. mpi4py can be used for sending higher-level data or a few big messages, while the C API can be used for sending C data or many small messages.

The same principle applies to multi-threaded code: It is possible, with some care, to start the threads through the Python API and then switch to e.g. native OS thread mutexes where any Python overhead would become too large. The resulting code would however be platform-specific as Windows and Unix-based systems have separate threading APIs.

Conclusions and future work

While the examples shown have been simple and in part contrived, they explore fundamental properties of loops and arrays that should apply in almost any real-world computation. For computations that can only be expressed as for-loops, and which is not available in a standard library, Cython should be a strong candidate. Certainly, for anything but very small amounts of data, a Python loop is unviable. The choice stands between Cython and other natively compiled technologies. Cython may not automatically produce quite as optimized code as e.g. Fortran, but we believe it is fast enough to still be attractive in many cases because of the high similarity with Python. With Cython and NumPy, copying of non-contiguous arrays is always explicit, which can be a huge advantage compared to some other technologies (like Fortran) when dealing with very large data sets.

[4]Finally, a different for loop syntax must be used, but this restriction will disappear in Cython 0.12.

[5]In addition to normal Python classes, Cython supports a type more efficient classes known as "extension types". For efficiency reasons these need explicit implementations provided for pickling and unpickling. See "Pickling and unpickling extension types" in [CPyPickle].

For the algorithms which are expressible as NumPy operations, the speedup is much lower, ranging from no speedup to around ten times. The Cython code is usually much more verbose and requires more decisions to be made at compile-time. Use of Cython in these situations seems much less clear cut. A good approach is to prototype using pure Python, and, if it is deemed too slow, optimize the important parts after benchmarks or code profiling.

Cython remains in active development. Because of the simple principles involved, new features are often easy to add, and are often the result of personal itch-scratching. Sometimes the experience has been that it is quicker to add a feature to Cython than to repeatedly write code to work around an issue. Some highlights of current development:

- Support for function-by-function profiling through the Python cProfile module was added in 0.11.3.
- Inner functions (closures) are maturing and will be released soon.
- Cython benefited from two Google Summer of Code [GSoC] projects over summer of 2009, which will result in better support for calling C++ and Fortran code.

One important and often requested feature for numerical users is template support. This would make it possible to make a single function support all array data types, without code duplication. Other possible features are improved parallel programming support, like OpenMP primitives. While no work is currently going on in these areas, the Cython developers remain conscious about these shortcomings.

One important future feature for numerical users is the new *memory view type*. K. W. Smith and D. S. Seljebotn started work on this in summer 2009 as part of Smith's Google Summer of Code.

The new Python buffer protocol based on PEP 3118 promise a shift in focus for array data. In Python 2.6 and greater, any Python object can export any array-like data to natively compiled code (like Cython code) in an efficient and standardized manner. In some respects this represents adoption of some NumPy functionality into the Python core. With this trend, it seems reasonable that Cython should provide good mechanisms for working with PEP 3118 buffers independently of NumPy. Incidentally, this will also provide a nice unified interface for interacting with C and Fortran arrays in various formats. Unlike NumPy, PEP 3118 buffers also supports pointer indirection-style arrays, sometimes used in C libraries.

With this new feature, the matrix multiplication routine could have been declared as:

```
def matmul(double[:,:] A,
           double[:,:] B,
           double[:,:] out):
    <...>
```

Using this syntax, a buffer is acquired from the arguments on entry. The interface of the argument variables are entirely decided by Cython, and it is not possible to use Python object operations. NumPy array methods like `A.mean()` will therefore no longer work. Instead, one will have to call `np.mean(A)` (which will work once NumPy supports PEP 3118). The advantage is that when Cython defines the interface, further optimizations can be introduced. Slices and arithmetic operations are not currently subject for optimization because of polymorphism. For instance, it would currently be impossible for Cython to optimize the multiplication operator, as it means different things for `ndarray` and its subclass `matrix`.

A nice benefit of the chosen syntax is that individual axis specifications becomes possible:

```
def matmul(double[:,::contig] A,
           double[::contig,:] B,
           double[:,:] out):
    <...>
```

Here, A is declared to have contiguous rows, without necessarily being contiguous as a whole. For instance, slicing along the first dimension (like `A[::3,:]`) would result in such an array. The `matmul` function can still benefit from the rows being declared as contiguous. Finally, we contemplate specific syntax for automatically making contiguous copies:

```
def matmul(in double[:,::contig] A,
           in double[::contig,:] B,
           double[:,:] out):
    <...>
```

This is particularly convenient when interfacing with C or Fortran libraries which demand such arrays.

Once basic memory access support is supported, it becomes possible to add optimizations for whole-array "vectorized" operations. For instance, this example could be supported:

```
cdef extern from "math.h":
    double sqrt(double)

def func(double[:] x, double[:] y):
    return sqrt(x**2 + y**2)
```

This would translate into a loop over the elements of `x` and `y`. Both a naive translation to C code, SSE code, GPU code, and use of BLAS or various C++ linear algebra libraries could eventually be supported. Whether Cython will actually move in this direction remains an open question. For now we simply note that even the most naive implementation of the above would lead to at least a 4 times speedup for large arrays over the corresponding NumPy expression; again due to NumPy's need for repeatedly transporting the data over the memory bus.

Acknowledgments

DSS wish to thank Stefan Behnel and Robert Bradshaw for enthusiastically sharing their Cython knowledge, and his advisor Hans Kristian Eriksen for helpful advice on writing this article. Cython is heavily based on [Pyrex] by Greg Ewing. The efficient NumPy array access was added to the language as the result of financial support by the Google Summer of Code program and Enthought Inc. The Centre of Mathematics for Applications at the University of Oslo and the Python Software Foundation provided funding to attend the SciPy '09 conference.

References

[Wilbers] I. M. Wilbers, H. P. Langtangen, Å. Ødegård, *Using Cython to Speed up Numerical Python Programs, Proceedings of MekIT'09, 2009.*

[Ramach] P. Ramachandran et al., *Performance of NumPy*, http://www.scipy.org/PerformancePython.

[Tutorial] S. Behnel, R. W. Bradshaw, D. S. Seljebotn, *Cython Tutorial*, Proceedings of the 8th Python in Science Conference, 2009.

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn, et al., *The Cython compiler*, http://cython.org.

[Pyrex] G. Ewing, *Pyrex: C-Extensions for Python*, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[Python] G. van Rossum et al., *The Python language*, http://python.org.

[NumPy] T. Oliphant, http://numpy.scipy.org

[SciPy] E. Jones, T. Oliphant, P. Peterson, http://scipy.org

[f2py] P. Peterson, *f2py: Fortran to Python interface generator*, http://scipy.org/F2py.

[Weave] E. Jones, *Weave: Tools for inlining C/C++ in Python*, http://scipy.org/Weave.

[Instant] M. Westlie, K. A. Mardal, M. S. Alnæs, *Instant: Inlining of C/C++ in Python* http://fenics.org/instant.

[Psyco] A. Rigo, *Psyco: Python Specializing Compiler*. http://psyco.sourceforge.net.

[Sage] W. Stein et al., *Sage Mathematics Software*, http://sagemath.org/.

[BLAS] L. S. Blackford, J. Demmel, et al., *An Updated Set of Basic Linear Algebra Subprograms (BLAS)*, ACM Trans. Math. Soft., 28-2 (2002), pp. 135--151. http://www.netlib.org/blas/

[ATLAS] C. Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, in Software: Practice and Experience, 5, 2, 101-121, 2005, http://math-atlas.sourceforge.net/

[GCC] The GNU C Compiler, http://gcc.gnu.org/.

[ICC] The Intel C Compiler, http://software.intel.com/en-us/intel-compilers/.

[mpi4py] L. Dalcin et al., *mpi4py: MPI bindings for Python*, http://code.google.com/p/mpi4py/.

[OpenMPI] Open MPI, http://open-mpi.org

[Docs] http://docs.cython.org

[directives] http://wiki.cython.org/enhancements/compilerdirectives

[CPyAPI] Python/C API Reference Manual, http://docs.python.org/c-api/

[CPyPickle] The Python pickle module. http://docs.python.org/library/pickle.html

[PEP3118] T. Oliphant, C. Banks, *PEP 3118 - Revising the buffer protocol*, http://www.python.org/dev/peps/pep-3118.

[GSoC] The Google Summer of Code Program, http://code.google.com/soc.