numexpr

Release 2.6.3.dev0

Dec 21, 2018

Contents

Contents:

CHAPTER 1

How it works

The string passed to `evaluate` is compiled into an object representing the expression and types of the arrays used by the function `numexpr`.

The expression is first compiled using Python's `compile` function (this means that the expressions have to be valid Python expressions). From this, the variable names can be taken. The expression is then evaluated using instances of a special object that keep track of what is being done to them, and which builds up the parse tree of the expression.

This parse tree is then compiled to a bytecode program, which describes how to perform the operation element-wise. The virtual machine uses "vector registers": each register is many elements wide (by default 4096 elements). The key to NumExpr's speed is handling chunks of elements at a time.

There are two extremes to evaluating an expression elementwise. You can do each operation as arrays, returning temporary arrays. This is what you do when you use NumPy: `2*a+3*b` uses three temporary arrays as large as `a` or `b`. This strategy wastes memory (a problem if your arrays are large), and also is not a good use of cache memory: for large arrays, the results of `2*a` and `3*b` won't be in cache when you do the add.

The other extreme is to loop over each element, as in:

```
for i in xrange(len(a)):
    c[i] = 2*a[i] + 3*b[i]
```

This doesn't consume extra memory, and is good for the cache, but, if the expression is not compiled to machine code, you will have a big case statement (or a bunch of if's) inside the loop, which adds a large overhead for each element, and will hurt the branch-prediction used on the CPU.

`numexpr` uses a in-between approach. Arrays are handled as chunks (of 4096 elements) at a time, using a register machine. As Python code, it looks something like this:

```
for i in xrange(0, len(a), 256):
   r0 = a[i:i+128]
   r1 = b[i:i+128]
   multiply(r0, 2, r2)
   multiply(r1, 3, r3)
   add(r2, r3, r2)
   c[i:i+128] = r2
```

(remember that the 3-arg form stores the result in the third argument, instead of allocating a new array). This achieves a good balance between cache and branch-prediction. And the virtual machine is written entirely in C, which makes it faster than the Python above. Furthermore the virtual machine is also multi-threaded, which allows for efficient parallelization of NumPy operations.

There is some more information and history at:

http://www.bitsofbits.com/2014/09/21/numpy-micro-optimization-and-numexpr/

CHAPTER 2

Expected performance

The range of speed-ups for NumExpr respect to NumPy can vary from 0.95x and 20x, being 2x, 3x or 4x typical values, depending on the complexity of the expression and the internal optimization of the operators used. The strided and unaligned case has been optimized too, so if the expression contains such arrays, the speed-up can increase significantly. Of course, you will need to operate with large arrays (typically larger than the cache size of your CPU) to see these improvements in performance.

Here there are some real timings. For the contiguous case:

```
In [1]: import numpy as np
In [2]: import numexpr as ne
In [3]: a = np.random.rand(1e6)
In [4]: b = np.random.rand(1e6)
In [5]: timeit 2*a + 3*b
10 loops, best of 3: 18.9 ms per loop
In [6]: timeit ne.evaluate("2*a + 3*b")
100 loops, best of 3: 5.83 ms per loop   # 3.2x: medium speed-up (simple expr)
In [7]: timeit 2*a + b**10
10 loops, best of 3: 158 ms per loop
In [8]: timeit ne.evaluate("2*a + b**10")
100 loops, best of 3: 7.59 ms per loop   # 20x: large speed-up due to optimised pow()
```

For unaligned arrays, the speed-ups can be even larger:

```
In [9]: a = np.empty(1e6, dtype="b1,f8")['f1']
In [10]: b = np.empty(1e6, dtype="b1,f8")['f1']
In [11]: a.flags.aligned, b.flags.aligned
Out[11]: (False, False)
In [12]: a[:] = np.random.rand(len(a))
In [13]: b[:] = np.random.rand(len(b))
In [14]: timeit 2*a + 3*b
10 loops, best of 3: 29.5 ms per loop
In [15]: timeit ne.evaluate("2*a + 3*b")
100 loops, best of 3: 7.46 ms per loop   # ~ 4x speed-up
```

CHAPTER 3

NumExpr 2.0 User Guide

The `numexpr` package supplies routines for the fast evaluation of array expressions elementwise by using a vector-based virtual machine.

Using it is simple:

```
>>> import numpy as np
>>> import numexpr as ne
>>> a = np.arange(10)
>>> b = np.arange(0, 20, 2)
>>> c = ne.evaluate("2*a+3*b")
>>> c
array([ 0,  8, 16, 24, 32, 40, 48, 56, 64, 72])
```

3.1 Building

NumExpr requires Python 2.6 or greater, and NumPy 1.7 or greater. It is built in the standard Python way:

```
$ python setup.py build
$ python setup.py install
```

You must have a C-compiler (i.e. MSVC on Windows and GCC on Linux) installed.

You can test `numexpr` with:

```
$ python -c "import numexpr; numexpr.test()"
```

3.2 Enabling Intel VML support

Starting from release 1.2 on, numexpr includes support for Intel's VML library. This allows for better performance on Intel architectures, mainly when evaluating transcendental functions (trigonometrical, exponential, ...). It also enables numexpr using several CPU cores.

If you have Intel's MKL (the library that embeds VML), just copy the `site.cfg.example` that comes in the distribution to `site.cfg` and edit the latter giving proper directions on how to find your MKL libraries in your system. After doing this, you can proceed with the usual building instructions listed above. Pay attention to the messages during the building process in order to know whether MKL has been detected or not. Finally, you can check the speed-ups on your machine by running the `bench/vml_timing.py` script (you can play with different parameters to the `set_vml_accuracy_mode()` and `set_vml_num_threads()` functions in the script so as to see how it would affect performance).

3.3 Usage Notes

NumExpr's principal routine is:

```
evaluate(ex, local_dict=None, global_dict=None, optimization='aggressive', truediv=
↪'auto')
```

where `ex` is a string forming an expression, like `"2*a+3*b"`. The values for `a` and `b` will by default be taken from the calling function's frame (through the use of `sys._getframe()`). Alternatively, they can be specified using the `local_dict` or `global_dict` arguments, or passed as keyword arguments.

The `optimization` parameter can take the values `'moderate'` or `'aggressive'`. `'moderate'` means that no optimization is made that can affect precision at all. `'aggressive'` (the default) means that the expression can be rewritten in a way that precision *could* be affected, but normally very little. For example, in `'aggressive'` mode, the transformation `x~**3` -> `x*x*x` is made, but not in `'moderate'` mode.

The *truediv* parameter specifies whether the division is a 'floor division' (False) or a 'true division' (True). The default is the value of *__future__.division* in the interpreter. See PEP 238 for details.

Expressions are cached, so reuse is fast. Arrays or scalars are allowed for the variables, which must be of type 8-bit boolean (bool), 32-bit signed integer (int), 64-bit signed integer (long), double-precision floating point number (float), 2x64-bit, double-precision complex number (complex) or raw string of bytes (str). If they are not in the previous set of types, they will be properly upcasted for internal use (the result will be affected as well). The arrays must all be the same size.

3.4 Datatypes supported internally

NumExpr operates internally only with the following types:

- 8-bit boolean (bool)
- 32-bit signed integer (int or int32)
- 64-bit signed integer (long or int64)
- 32-bit single-precision floating point number (float or float32)
- 64-bit, double-precision floating point number (double or float64)
- 2x64-bit, double-precision complex number (complex or complex128)
- Raw string of bytes (str)

If the arrays in the expression does not match any of these types, they will be upcasted to one of the above types (following the usual type inference rules, see below). Have this in mind when doing estimations about the memory consumption during the computation of your expressions.

Also, the types in NumExpr conditions are somewhat stricter than those of Python. For instance, the only valid constants for booleans are `True` and `False`, and they are never automatically cast to integers.

3.5 Casting rules

Casting rules in NumExpr follow closely those of *NumPy*. However, for implementation reasons, there are some known exceptions to this rule, namely:

- When an array with type `int8`, `uint8`, `int16` or `uint16` is used inside NumExpr, it is internally upcasted to an `int` (or `int32` in NumPy notation).
- When an array with type `uint32` is used inside NumExpr, it is internally upcasted to a `long` (or `int64` in NumPy notation).
- A floating point function (e.g. `sin`) acting on `int8` or `int16` types returns a `float64` type, instead of the `float32` that is returned by NumPy functions. This is mainly due to the absence of native `int8` or `int16` types in NumExpr.
- In operations implying a scalar and an array, the normal rules of casting are used in NumExpr, in contrast with NumPy, where array types takes priority. For example, if `a` is an array of type `float32` and `b` is an scalar of type `float64` (or Python `float` type, which is equivalent), then `a*b` returns a `float64` in NumExpr, but a `float32` in NumPy (i.e. array operands take priority in determining the result type). If you need to keep the result a `float32`, be sure you use a `float32` scalar too.

3.6 Supported operators

NumExpr supports the set of operators listed below:

- Logical operators: `&, |, ~`
- Comparison operators: `<, <=, ==, !=, >=, >`
- Unary arithmetic operators: `-`
- Binary arithmetic operators: `+, -, *, /, **, %, <<, >>`

3.7 Supported functions

The next are the current supported set:

- `where(bool, number1, number2)`: `number` – number1 if the bool condition is true, number2 otherwise.
- `{sin,cos,tan}(float|complex)`: `float|complex` – trigonometric sine, cosine or tangent.
- `{arcsin,arccos,arctan}(float|complex)`: `float|complex` – trigonometric inverse sine, cosine or tangent.
- `arctan2(float1, float2)`: `float` – trigonometric inverse tangent of float1/float2.
- `{sinh,cosh,tanh}(float|complex)`: `float|complex` – hyperbolic sine, cosine or tangent.
- `{arcsinh,arccosh,arctanh}(float|complex)`: `float|complex` – hyperbolic inverse sine, cosine or tangent.
- `{log,log10,log1p}(float|complex)`: `float|complex` – natural, base-10 and log(1+x) logarithms.
- `{exp,expm1}(float|complex)`: `float|complex` – exponential and exponential minus one.
- `sqrt(float|complex)`: `float|complex` – square root.

- `abs(float|complex)`: `float|complex` – absolute value.
- `conj(complex)`: `complex` – conjugate value.
- `{real,imag}(complex)`: `float` – real or imaginary part of complex.
- `complex(float, float)`: `complex` – complex from real and imaginary parts.
- `contains(str, str)`: `bool` – returns True for every string in `op1` that contains `op2`.

3.8 Notes

- `abs()` for complex inputs returns a `complex` output too. This is a departure from NumPy where a `float` is returned instead. However, NumExpr is not flexible enough yet so as to allow this to happen. Meanwhile, if you want to mimic NumPy behaviour, you may want to select the real part via the `real` function (e.g. `real(abs(cplx))`) or via the `real` selector (e.g. `abs(cplx).real`).

More functions can be added if you need them. Note however that NumExpr 2.6 is in maintenance mode and a new major revision is under development.

3.9 Supported reduction operations

The next are the current supported set:

- `sum(number, axis=None)`: Sum of array elements over a given axis. Negative axis are not supported.
- `prod(number, axis=None)`: Product of array elements over a given axis. Negative axis are not supported.

Note: because of internal limitations, reduction operations must appear the last in the stack. If not, it will be issued an error like:

```
>>> ne.evaluate('sum(1)*(-1)')
RuntimeError: invalid program: reduction operations must occur last
```

3.10 General routines

- `evaluate(expression, local_dict=None, global_dict=None, optimization='aggressive', truediv='auto')`: Evaluate a simple array expression element-wise. See examples above.
- `re_evaluate(local_dict=None)`: Re-evaluate the last array expression without any check. This is meant for accelerating loops that are re-evaluating the same expression repeatedly without changing anything else than the operands. If unsure, use evaluate() which is safer.
- `test()`: Run all the tests in the test suite.
- `print_versions()`: Print the versions of software that numexpr relies on.
- `set_num_threads(nthreads)`: Sets a number of threads to be used in operations. Returns the previous setting for the number of threads. See note below to see how the number of threads is set via environment variables.

  If you are using VML, you may want to use *set_vml_num_threads(nthreads)* to perform the parallel job with VML instead. However, you should get very similar performance with VML-optimized functions, and VML's parallelizer cannot deal with common expressions like *(x+1)*(x-2)*, while NumExpr's one can.

- `detect_number_of_cores()`: Detects the number of cores on a system.

Note on the maximum number of threads: Threads are spawned at import-time, with the number being set by the environment variable `NUMEXPR_MAX_THREADS`. Example:

```
import os; os.environ['NUMEXPR_MAX_THREADS'] = '16'
```

The default maximum thread count is 64. The initial number of threads _that are **used**_ will be set to the number of cores detected in the system or 8, whichever is **lower**. For historical reasons, the `NUMEXPR_NUM_THREADS` environment variable is also honored at initialization time and, if defined, the initial number of threads will be set to this value instead. Alternatively, the *OMP_NUM_THREADS* environment variable is also honored, but beware because that might affect to other OpenMP applications too.

3.11 Intel's VML specific support routines

When compiled with Intel's VML (Vector Math Library), you will be able to use some additional functions for controlling its use. These are:

- `set_vml_accuracy_mode(mode)`: Set the accuracy for VML operations.

  The `mode` parameter can take the values:

  - `'low'`: Equivalent to VML_LA - low accuracy VML functions are called
  - `'high'`: Equivalent to VML_HA - high accuracy VML functions are called
  - `'fast'`: Equivalent to VML_EP - enhanced performance VML functions are called

  It returns the previous mode.

  This call is equivalent to the `vmlSetMode()` in the VML library. See:

  http://www.intel.com/software/products/mkl/docs/webhelp/vml/vml_DataTypesAccuracyModes.html

  for more info on the accuracy modes.

- `set_vml_num_threads(nthreads)`: Suggests a maximum number of threads to be used in VML operations.

  This function is equivalent to the call `mkl_domain_set_num_threads(nthreads, MKL_VML)` in the MKL library. See:

  http://www.intel.com/software/products/mkl/docs/webhelp/support/functn_mkl_domain_set_num_threads.html

  for more info about it.

- `get_vml_version()`: Get the VML/MKL library version.

3.12 Authors

Numexpr was initially written by David Cooke, and extended to more types by Tim Hochberg.

Francesc Alted contributed support for booleans and simple-precision floating point types, efficient strided and unaligned array operations and multi-threading code.

Ivan Vilata contributed support for strings.

Gregor Thalhammer implemented the support for Intel VML (Vector Math Library).

Mark Wiebe added support for the new iterator in NumPy, which allows for better performance in more scenarios (like broadcasting, fortran-ordered or non-native byte orderings).

Gaëtan de Menten contributed important bug fixes and speed enhancements.

Antonio Valentino contributed the port to Python 3.

Google Inc. contributed bug fixes.

David Cox improved readability of the Readme.

Robert A. McLeod contributed bug fixes and ported the documentation to numexpr.readthedocs.io.

3.13 License

NumExpr is distributed under the MIT license.

CHAPTER 4

Performance of the Virtual Machine in NumExpr2.0

Numexpr 2.0 leverages a new virtual machine completely based on the new ndarray iterator introduced in NumPy 1.6. This represents a nice combination of the advantages of using the new iterator, while retaining the ability to avoid copies in memory as well as the multi-threading capabilities of the previous virtual machine (1.x series).

The increased performance of the new virtual machine can be seen in several scenarios, like:

- *Broadcasting*. Expressions containing arrays that needs to be broadcasted, will not need additional memory (i.e. they will be broadcasted on-the-fly).
- *Non-native dtypes*. These will be translated to native dtypes on-the-fly, so there is not need to convert the whole arrays first.
- *Fortran-ordered arrays*. The new iterator will find the best path to optimize operations on such arrays, without the need to transpose them first.

There is a drawback though: performance with small arrays suffers a bit because of higher set-up times for the new virtual machine. See below for detailed benchmarks.

4.1 Some benchmarks for best-case scenarios

Here you have some benchmarks of some scenarios where the new virtual machine actually represents an advantage in terms of speed (also memory, but this is not shown here). As you will see, the improvement is notable in many areas, ranging from 3x to 6x faster operations.

4.1.1 Broadcasting

```
>>> a = np.arange(1e3)
>>> b = np.arange(1e6).reshape(1e3, 1e3)
```

```
>>> timeit ne.evaluate("a*(b+1)")    # 1.4.2
100 loops, best of 3: 16.4 ms per loop
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 2.0
100 loops, best of 3: 5.2 ms per loop
```

4.1.2 Non-native types

```
>>> a = np.arange(1e6, dtype=">f8")
>>> b = np.arange(1e6, dtype=">f8")
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 1.4.2
100 loops, best of 3: 17.2 ms per loop
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 2.0
100 loops, best of 3: 6.32 ms per loop
```

4.1.3 Fortran-ordered arrays

```
>>> a = np.arange(1e6).reshape(1e3, 1e3).copy('F')
>>> b = np.arange(1e6).reshape(1e3, 1e3).copy('F')
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 1.4.2
10 loops, best of 3: 32.8 ms per loop
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 2.0
100 loops, best of 3: 5.62 ms per loop
```

4.1.4 Mix of 'non-native' arrays, Fortran-ordered, and using broadcasting

```
>>> a = np.arange(1e3, dtype='>f8').copy('F')
>>> b = np.arange(1e6, dtype='>f8').reshape(1e3, 1e3).copy('F')
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 1.4.2
10 loops, best of 3: 21.2 ms per loop
```

```
>>> timeit ne.evaluate("a*(b+1)")  # 2.0
100 loops, best of 3: 5.22 ms per loop
```

4.1.5 Longer setup-time

The only drawback of the new virtual machine is during the computation of small arrays:

```
>>> a = np.arange(10)
>>> b = np.arange(10)

>>> timeit ne.evaluate("a*(b+1)")  # 1.4.2
10000 loops, best of 3: 22.1 µs per loop

>>> timeit ne.evaluate("a*(b+1)")  # 2.0
10000 loops, best of 3: 30.6 µs per loop
```

i.e. the new virtual machine takes a bit more time to set-up (around 8 µs in this machine). However, this should be not too important because for such a small arrays NumPy is always a better option:

```
>>> timeit c = a*(b+1)
100000 loops, best of 3: 4.16 µs per loop
```

And for arrays large enough the difference is negligible:

```
>>> a = np.arange(1e6)
>>> b = np.arange(1e6)

>>> timeit ne.evaluate("a*(b+1)")   # 1.4.2
100 loops, best of 3: 5.77 ms per loop

>>> timeit ne.evaluate("a*(b+1)")   # 2.0
100 loops, best of 3: 5.77 ms per loop
```

4.2 Conclusion

The new virtual machine introduced in numexpr 2.0 brings more performance in many different scenarios (broadcast, non-native dtypes, Fortran-orderd arrays), while it shows slightly worse performance for small arrays. However, as numexpr is more geared to compute large arrays, the new virtual machine should be good news for numexpr users in general.

CHAPTER 5

NumExpr with Intel MKL

Numexpr has support for Intel's VML (included in Intel's MKL) in order to accelerate the evaluation of transcendental functions on Intel CPUs. Here it is a small example on the kind of improvement you may get by using it.

5.1 A first benchmark

Firstly, we are going to exercise how MKL performs when computing a couple of simple expressions. One is a pure algebraic one: `2*y + 4*x` and the other contains transcendental functions: `sin(x)**2 + cos(y)**2`.

For this, we are going to use this worksheet. I (Francesc Alted) ran this benchmark on a Intel Xeon E3-1245 v5 @ 3.50GHz. Here are the results when not using MKL:

```
NumPy version: 1.11.1
Time for an algebraic expression:     0.168 s / 6.641 GB/s
Time for a transcendental expression: 1.945 s / 0.575 GB/s
Numexpr version: 2.6.1. Using MKL: False
Time for an algebraic expression:     0.058 s / 19.116 GB/s
Time for a transcendental expression: 0.283 s / 3.950 GB/s
```

And now, using MKL:

```
NumPy version: 1.11.1
Time for an algebraic expression:     0.169 s / 6.606 GB/s
Time for a transcendental expression: 1.943 s / 0.575 GB/s
Numexpr version: 2.6.1. Using MKL: True
Time for an algebraic expression:     0.058 s / 19.153 GB/s
Time for a transcendental expression: 0.075 s / 14.975 GB/s
```

As you can see, numexpr using MKL can be up to 3.8x faster for the case of the transcendental expression. Also, you can notice that the pure algebraic expression is not accelerated at all. This is completely expected, as the MKL is offering accelerations for CPU bounded functions (sin, cos, tan, exp, log, sinh...) and not pure multiplications or adds.

Finally, note how numexpr+MKL can be up to 26x faster than using a pure NumPy solution. And this was using a processor with just four physical cores; you should expect more speedup as you throw more cores at that.

5.2 More benchmarks (older)

Numexpr & VML can both use several threads for doing computations. Let's see how performance improves by using 1 or 2 threads on a 2-core Intel CPU (Core2 E8400 @ 3.00GHz).

5.2.1 Using 1 thread

Here we have some benchmarks on the improvement of speed that Intel's VML can achieve. First, look at times by some easy expression containing sine and cosine operations *without* using VML:

```
In [17]: ne.use_vml
Out[17]: False

In [18]: x = np.linspace(-1, 1, 1e6)

In [19]: timeit np.sin(x)**2+np.cos(x)**2
10 loops, best of 3: 43.1 ms per loop

In [20]: ne.set_num_threads(1)
Out[20]: 2

In [21]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
10 loops, best of 3: 29.5 ms per loop
```

and now using VML:

```
In [37]: ne.use_vml
Out[37]: True

In [38]: x = np.linspace(-1, 1, 1e6)

In [39]: timeit np.sin(x)**2+np.cos(x)**2
10 loops, best of 3: 42.8 ms per loop

In [40]: ne.set_num_threads(1)
Out[40]: 2

In [41]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
100 loops, best of 3: 19.8 ms per loop
```

Hey, VML can accelerate computations by a 50% using a single CPU. That's great!

5.2.2 Using 2 threads

First, look at the time of the non-VML numexpr when using 2 threads:

```
In [22]: ne.set_num_threads(2)
Out[22]: 1
```

(continues on next page)

(continued from previous page)

```
In [23]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
100 loops, best of 3: 15.3 ms per loop
```

OK. We've got an almost perfect 2x improvement in speed with regard to the 1 thread case. Let's see about the VML-powered numexpr version:

```
In [43]: ne.set_num_threads(2)
Out[43]: 1

In [44]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
100 loops, best of 3: 12.2 ms per loop
```

Ok, that's about 1.6x improvement over the 1 thread VML computation, and still a 25% of improvement over the non-VML version. Good, native numexpr multithreading code really looks very efficient!

5.2.3 Numexpr native threading code vs VML's one

You may already know that both numexpr and Intel's VML do have support for multithreaded computations, but you might be curious about which one is more efficient, so here it goes a hint. First, using the VML multithreaded implementation:

```
In [49]: ne.set_vml_num_threads(2)

In [50]: ne.set_num_threads(1)
Out[50]: 1

In [51]: ne.set_vml_num_threads(2)

In [52]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
100 loops, best of 3: 16.8 ms per loop
```

and now, using the native numexpr threading code:

```
In [53]: ne.set_num_threads(2)
Out[53]: 1

In [54]: ne.set_vml_num_threads(1)

In [55]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
100 loops, best of 3: 12 ms per loop
```

This means that numexpr's native multithreaded code is about 40% faster than VML's for this case. So, in general, you should use the former with numexpr (and this is the default actually).

5.2.4 Mixing numexpr's and VML multithreading capabilities

Finally, you might be tempted to use both multithreading codes at the same time, but you will be deceived about the improvement in performance:

```
In [57]: ne.set_vml_num_threads(2)

In [58]: timeit ne.evaluate('sin(x)**2+cos(x)**2')
100 loops, best of 3: 17.7 ms per loop
```

Your code actually performs much worse. That's normal too because you are trying to run 4 threads on a 2-core CPU. For CPUs with many cores, you may want to try with different threading configurations, but as a rule of thumb, numexpr's one will generally win.

CHAPTER 6

NumExpr API

Numexpr is a fast numerical expression evaluator for NumPy. With it, expressions that operate on arrays (like "3*a+4*b") are accelerated and use less memory than doing the same calculation in Python.

See:

https://github.com/pydata/numexpr

for more info about it.

numexpr.**evaluate**(*ex*, *local_dict=None*, *global_dict=None*, *out=None*, *order='K'*, *casting='safe'*, ***kwargs*)

Evaluate a simple array expression element-wise, using the new iterator.

ex is a string forming an expression, like "2*a+3*b". The values for "a" and "b" will by default be taken from the calling function's frame (through use of sys._getframe()). Alternatively, they can be specifed using the 'local_dict' or 'global_dict' arguments.

Parameters

local_dict [dictionary, optional] A dictionary that replaces the local operands in current frame.

global_dict [dictionary, optional] A dictionary that replaces the global operands in current frame.

out [NumPy array, optional] An existing array where the outcome is going to be stored. Care is required so that this array has the same shape and type than the actual outcome of the computation. Useful for avoiding unnecessary new array allocations.

order [{'C', 'F', 'A', or 'K'}, optional] Controls the iteration order for operands. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. For efficient computations, typically 'K'eep order (the default) is desired.

casting [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur when making a copy or buffering. Setting this to 'unsafe' is not recommended, as it can adversely affect accumulations.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

numexpr.**re_evaluate**(*local_dict=None*)
Re-evaluate the previous executed array expression without any check.

This is meant for accelerating loops that are re-evaluating the same expression repeatedly without changing anything else than the operands. If unsure, use evaluate() which is safer.

Parameters

local_dict [dictionary, optional] A dictionary that replaces the local operands in current frame.

numexpr.**disassemble**(*nex*)
Given a NumExpr object, return a list which is the program disassembled.

numexpr.**NumExpr**(*ex*, *signature=()*, ***kwargs*)
Compile an expression built using E.<variable> variables to a function.

ex can also be specified as a string "2*a+3*b".

The order of the input variables and their types can be specified using the signature parameter, which is a list of (name, type) pairs.

Returns a *NumExpr* object containing the compiled function.

numexpr.**get_vml_version**()
Get the VML/MKL library version.

numexpr.**set_vml_accuracy_mode**(*mode*)
Set the accuracy mode for VML operations.

The *mode* parameter can take the values: - 'high': high accuracy mode (HA), <1 least significant bit - 'low': low accuracy mode (LA), typically 1-2 least significant bits - 'fast': enhanced performance mode (EP) - None: mode settings are ignored

This call is equivalent to the *vmlSetMode()* in the VML library. See:

http://www.intel.com/software/products/mkl/docs/webhelp/vml/vml_DataTypesAccuracyModes.html

for more info on the accuracy modes.

Returns old accuracy settings.

numexpr.**set_vml_num_threads**(*nthreads*)
Suggests a maximum number of threads to be used in VML operations.

This function is equivalent to the call *mkl_domain_set_num_threads(nthreads, MKL_DOMAIN_VML)* in the MKL library. See:

http://www.intel.com/software/products/mkl/docs/webhelp/support/functn_mkl_domain_set_num_threads.html

for more info about it.

numexpr.**set_num_threads**(*nthreads*)
Sets a number of threads to be used in operations.

Returns the previous setting for the number of threads.

During initialization time Numexpr sets this number to the number of detected cores in the system (see *detect_number_of_cores()*).

If you are using Intel's VML, you may want to use *set_vml_num_threads(nthreads)* to perform the parallel job with VML instead. However, you should get very similar performance with VML-optimized functions, and VML's parallelizer cannot deal with common expresions like *(x+1)*(x-2)*, while Numexpr's one can.

numexpr.**detect_number_of_cores**()
Detects the number of cores on a system. Cribbed from pp.

numexpr.**detect_number_of_threads**()
If this is modified, please update the note in: https://github.com/pydata/numexpr/wiki/Numexpr-Users-Guide

numexpr.**ncores**
The number of (virtual) cores detected.

numexpr.**nthreads**
The number of threads currently in-use.

numexpr.**MAX_THREADS**
The maximum number of threads, as set by the environment variable NUMEXPR_MAX_THREADS

numexpr.**version**
The version of NumExpr.

6.1 Tests submodule

numexpr.tests.**test**(*verbosity=1*)
Run all the tests in the test suite.

numexpr.tests.**print_versions**()
Print the versions of software that numexpr relies on.

CHAPTER 7

Release Notes

7.1 Release notes for Numexpr 2.6 series

7.1.1 Changes from 2.6.9 to 2.6.10

- #XXX version-specific blurb XXX#

7.1.2 Changes from 2.6.8 to 2.6.9

- Thanks to Mike Toews for more robust handling of the thread-setting environment variables.
- With Appveyor updating to Python 3.7.1, wheels for Python 3.7 are now available in addition to those for other OSes.

7.1.3 Changes from 2.6.7 to 2.6.8

- Add check to make sure that *f_locals* is not actually *f_globals* when we do the *f_locals* clear to avoid the #310 memory leak issue.
- Compare NumPy versions using *distutils.version.LooseVersion* to avoid issue #312 when working with NumPy development versions.
- As part of *multibuild*, wheels for Python 3.7 for Linux and MacOSX are now available on PyPI.

7.1.4 Changes from 2.6.6 to 2.6.7

- Thanks to Lehman Garrison for finding and fixing a bug that exhibited memory leak-like behavior. The use in *numexpr.evaluate* of *sys._getframe* combined with *.f_locals* from that frame object results an extra refcount on objects in the frame that calls *numexpr.evaluate*, and not *evaluate*'s frame. So if the calling frame remains in scope for a long time (such as a procedural script where *numexpr* is called from the base frame) garbage collection would never occur.

- Imports for the *numexpr.test* submodule were made lazy in the *numexpr* module.

7.1.5 Changes from 2.6.5 to 2.6.6

- Thanks to Mark Dickinson for a fix to the thread barrier that occassionally suffered from spurious wakeups on MacOSX.

7.1.6 Changes from 2.6.4 to 2.6.5

- The maximum thread count can now be set at import-time by setting the environment variable 'NUMEXPR_MAX_THREADS'. The default number of max threads was lowered from 4096 (which was deemed excessive) to 64.
- A number of imports were removed (pkg_resources) or made lazy (cpuinfo) in order to speed load-times for downstream packages (such as *pandas*, *sympy*, and *tables*). Import time has dropped from about 330 ms to 90 ms. Thanks to Jason Sachs for pointing out the source of the slow-down.
- Thanks to Alvaro Lopez Ortega for updates to benchmarks to be compatible with Python 3.
- Travis and AppVeyor now fail if the test module fails or errors.
- Thanks to Mahdi Ben Jelloul for a patch that removed a bug where constants in *where* calls would raise a ValueError.
- Fixed a bug whereby all-constant power operations would lead to infinite recursion.

7.1.7 Changes from 2.6.3 to 2.6.4

- Christoph Gohlke noticed a lack of coverage for the 2.6.3 *floor* and *ceil* functions for MKL that caused seg-faults in test, so thanks to him for that.

7.1.8 Changes from 2.6.2 to 2.6.3

- Documentation now available at readthedocs.io.
- Support for floor() and ceil() functions added by Caleb P. Burns.
- NumPy requirement increased from 1.6 to 1.7 due to changes in iterator flags (#245).
- Sphinx autodocs support added for documentation on readthedocs.org.
- Fixed a bug where complex constants would return an error, fixing problems with *sympy* when using NumExpr as a backend.
- Fix for #277 whereby arrays of shape (1,...) would be reduced as if they were full reduction. Behavoir now matches that of NumPy.
- String literals are automatically encoded into 'ascii' bytes for convience (see #281).

7.1.9 Changes from 2.6.1 to 2.6.2

- Updates to keep with API changes in newer NumPy versions (#228). Thanks to Oleksandr Pavlyk.
- Removed several warnings (#226 and #227). Thanks to Oleksander Pavlyk.
- Fix bugs in function *stringcontains()* (#230). Thanks to Alexander Shadchin.

- Detection of the POWER processor (#232). Thanks to Breno Leitao.
- Fix pow result casting (#235). Thanks to Fernando Seiti Furusato.
- Fix integers to negative integer powers (#240). Thanks to Antonio Valentino.
- Detect numpy exceptions in expression evaluation (#240). Thanks to Antonio Valentino.
- Better handling of RC versions (#243). Thanks to Antonio Valentino.

7.1.10 Changes from 2.6.0 to 2.6.1

- Fixed a performance regression in some situations as consequence of increasing too much the BLOCK_SIZE1 constant. After more careful benchmarks (both in VML and non-VML modes), the value has been set again to 1024 (down from 8192). The benchmarks have been made with a relatively new processor (Intel Xeon E3-1245 v5 @ 3.50GHz), so they should work well for a good range of processors again.
- Added NetBSD support to CPU detection. Thanks to Thomas Klausner.

7.1.11 Changes from 2.5.2 to 2.6.0

- Introduced a new re_evaluate() function for re-evaluating the previous executed array expression without any check. This is meant for accelerating loops that are re-evaluating the same expression repeatedly without changing anything else than the operands. If unsure, use evaluate() which is safer.
- The BLOCK_SIZE1 and BLOCK_SIZE2 constants have been re-checked in order to find a value maximizing most of the benchmarks in bench/ directory. The new values (8192 and 16 respectively) give somewhat better results (~5%) overall. The CPU used for fine tuning is a relatively new Haswell processor (E3-1240 v3).
- The '–name' flag for *setup.py* returning the name of the package is honored now (issue #215).

7.1.12 Changes from 2.5.1 to 2.5.2

- conj() and abs() actually added as VML-powered functions, preventing the same problems than log10() before (PR #212). Thanks to Tom Kooij for the fix!

7.1.13 Changes from 2.5 to 2.5.1

- Fix for log10() and conj() functions. These produced wrong results when numexpr was compiled with Intel's MKL (which is a popular build since Anaconda ships it by default) and non-contiguous data (issue #210). Thanks to Arne de Laat and Tom Kooij for reporting and providing a nice test unit.
- Fix that allows numexpr-powered apps to be profiled with pympler. Thanks to @nbecker.

7.1.14 Changes from 2.4.6 to 2.5

- Added locking for allowing the use of numexpr in multi-threaded callers (this does not prevent numexpr to use multiple cores simultaneously). (PR #199, Antoine Pitrou, PR #200, Jenn Olsen).
- Added new min() and max() functions (PR #195, CJ Carey).

7.1.15 Changes from 2.4.5 to 2.4.6

- Fixed some UserWarnings in Solaris (PR #189, Graham Jones).
- Better handling of MSVC defines. (#168, Francesc Alted).

7.1.16 Changes from 2.4.4 to 2.4.5

- Undone a 'fix' for a harmless data race. (#185 Benedikt Reinartz, Francesc Alted).
- Ignore NumPy warnings (overflow/underflow, divide by zero and others) that only show up in Python3. Masking these warnings in tests is fine because all the results are checked to be valid. (#183, Francesc Alted).

7.1.17 Changes from 2.4.3 to 2.4.4

- Fix bad #ifdef for including stdint on Windows (PR #186, Mike Sarahan).

7.1.18 Changes from 2.4.3 to 2.4.4

- Honor OMP_NUM_THREADS as a fallback in case NUMEXPR_NUM_THREADS is not set. Fixes #161. (PR #175, Stefan Erb).
- Added support for AppVeyor (PR #178 Andrea Bedini)
- Fix to allow numexpr to be imported after eventlet.monkey_patch(), as suggested in #118 (PR #180 Ben Moran).
- Fix harmless data race that triggers false positives in ThreadSanitizer. (PR #179, Clement Courbet).
- Fixed some string tests on Python 3 (PR #182, Antonio Valentino).

7.1.19 Changes from 2.4.2 to 2.4.3

- Comparisons with empty strings work correctly now. Fixes #121 and PyTables #184.

7.1.20 Changes from 2.4.1 to 2.4.2

- Improved setup.py so that pip can query the name and version without actually doing the installation. Thanks to Joris Borgdorff.

7.1.21 Changes from 2.4 to 2.4.1

- Added more configuration examples for compiling with MKL/VML support. Thanks to Davide Del Vento.
- Symbol MKL_VML changed into MKL_DOMAIN_VML because the former is deprecated in newer MKL. Thanks to Nick Papior Andersen.
- Better determination of methods in *cpuinfo* module. Thanks to Marc Jofre.
- Improved NumPy version determination (handy for 1.10.0). Thanks to Åsmund Hjulstad.
- Benchmarks run now with both Python 2 and Python 3. Thanks to Zoran Plesivčak.

7.1.22 Changes from 2.3.1 to 2.4

- A new *contains()* function has been added for detecting substrings in strings. Only plain strings (bytes) are supported for now. See PR #135 and ticket #142. Thanks to Marcin Krol.
- New version of setup.py that allows better management of NumPy dependency. See PR #133. Thanks to Aleks Bunin.

7.1.23 Changes from 2.3 to 2.3.1

- Added support for shift-left (<<) and shift-right (>>) binary operators. See PR #131. Thanks to fish2000!
- Removed the rpath flag for the GCC linker, because it is probably not necessary and it chokes to clang.

7.1.24 Changes from 2.2.2 to 2.3

- Site has been migrated to https://github.com/pydata/numexpr. All new tickets and PR should be directed there.
- [ENH] A *conj()* function for computing the conjugate of complex arrays has been added. Thanks to David Menéndez. See PR #125.
- [FIX] Fixed a DeprecationWarning derived of using oa_ndim – 0 and op_axes – NULL when using NpyIter_AdvancedNew() and NumPy 1.8. Thanks to Mark Wiebe for advise on how to fix this properly.

7.1.25 Changes from 2.2.1 to 2.2.2

- The *copy_args* argument of *NumExpr* function has been brought lack. This has been mainly necessary for compatibility with *PyTables < 3.0*, which I decided to continue to support. Fixed #115.
- The *__nonzero__* method in *ExpressionNode* class has been commented out. This is also for compatibility with *PyTables < 3.0*. See #24 for details.
- Fixed the type of some parameters in the C extension so that s390 architecture compiles. Fixes #116. Thank to Antonio Valentino for reporting and the patch.

7.1.26 Changes from 2.2 to 2.2.1

- Fixes a secondary effect of "from numpy.testing import *", where division is imported now too, so only then necessary functions from there are imported now. Thanks to Christoph Gohlke for the patch.

7.1.27 Changes from 2.1 to 2.2

- [LICENSE] Fixed a problem with the license of the numexpr/win32/pthread.{c,h} files emulating pthreads on Windows platforms. After persmission from the original authors is granted, these files adopt the MIT license and can be redistributed without problems. See issue #109 for details (https://code.google.com/p/numexpr/issues/detail?id-110).
- [ENH] Improved the algorithm to decide the initial number of threads to be used. This was necessary because by default, numexpr was using a number of threads equal to the detected number of cores, and this can be just too much for moder systems where this number can be too high (and counterporductive for performance in many cases). Now, the 'NUMEXPR_NUM_THREADS' environment variable is honored, and in

case this is not present, a maximum number of *8* threads are setup initially. The new algorithm is fully described in the Users Guide now in the note of 'General routines' section: https://code.google.com/p/numexpr/wiki/UsersGuide#General_routines. Closes #110.

- [ENH] numexpr.test() returns *TestResult* instead of None now. Closes #111.
- [FIX] Modulus with zero with integers no longer crashes the interpreter. It nows puts a zero in the result. Fixes #107.
- [API CLEAN] Removed *copy_args* argument of *evaluate*. This should only be used by old versions of PyTables (< 3.0).
- [DOC] Documented the *optimization* and *truediv* flags of *evaluate* in Users Guide (https://code.google.com/p/numexpr/wiki/UsersGuide).

7.1.28 Changes from 2.0.1 to 2.1

- Dropped compatibility with Python < 2.6.
- Improve compatibiity with Python 3:
  - switch from PyString to PyBytes API (requires Python >- 2.6).
  - fixed incompatibilities regarding the int/long API
  - use the Py_TYPE macro
  - use the PyVarObject_HEAD_INIT macro instead of PyObject_HEAD_INIT
- Fixed several issues with different platforms not supporting multithreading or subprocess properly (see tickets #75 and #77).
- Now, when trying to use pure Python boolean operators, 'and', 'or' and 'not', an error is issued suggesting that '&', '|' and '~' should be used instead (fixes #24).

7.1.29 Changes from 2.0 to 2.0.1

- Added compatibility with Python 2.5 (2.4 is definitely not supported anymore).
- *numexpr.evaluate* is fully documented now, in particular the new *out*, *order* and *casting* parameters.
- Reduction operations are fully documented now.
- Negative axis in reductions are not supported (they have never been actually), and a *ValueError* will be raised if they are used.

7.1.30 Changes from 1.x series to 2.0

- Added support for the new iterator object in NumPy 1.6 and later.

  This allows for better performance with operations that implies broadcast operations, fortran-ordered or non-native byte orderings. Performance for other scenarios is preserved (except for very small arrays).
- Division in numexpr is consistent now with Python/NumPy. Fixes #22 and #58.
- Constants like "2." or "2.0" must be evaluated as float, not integer. Fixes #59.
- *evaluate()* function has received a new parameter *out* for storing the result in already allocated arrays. This is very useful when dealing with large arrays, and a allocating new space for keeping the result is not acceptable. Closes #56.

- Maximum number of threads raised from 256 to 4096. Machines with a higher number of cores will still be able to import numexpr, but limited to 4096 (which is an absurdly high number already).

7.1.31 Changes from 1.4.1 to 1.4.2

- Multithreaded operation is disabled for small arrays (< 32 KB). This allows to remove the overhead of multithreading for such a small arrays. Closes #36.
- Dividing int arrays by zero gives a 0 as result now (and not a floating point exception anymore. This behaviour mimics NumPy. Thanks to Gaëtan de Menten for the fix. Closes #37.
- When compiled with VML support, the number of threads is set to 1 for VML core, and to the number of cores for the native pthreads implementation. This leads to much better performance. Closes #39.
- Fixed different issues with reduction operations (*sum*, *prod*). The problem is that the threaded code does not work well for broadcasting or reduction operations. Now, the serial code is used in those cases. Closes #41.
- Optimization of "compilation phase" through a better hash. This can lead up to a 25% of improvement when operating with variable expressions over small arrays. Thanks to Gaëtan de Menten for the patch. Closes #43.
- The `set_num_threads` now returns the number of previous thread setting, as stated in the docstrings.

7.1.32 Changes from 1.4 to 1.4.1

- Mingw32 can also work with pthreads compatibility code for win32. Fixes #31.
- Fixed a problem that used to happen when running Numexpr with threads in subprocesses. It seems that threads needs to be initialized whenever a subprocess is created. Fixes #33.
- The GIL (Global Interpreter Lock) is released during computations. This should allow for better resource usage for multithreaded apps. Fixes #35.

7.1.33 Changes from 1.3.1 to 1.4

- Added support for multi-threading in pure C. This is to avoid the GIL and allows to squeeze the best performance in both multi-core machines.
- David Cooke contributed a thorough refactorization of the opcode machinery for the virtual machine. With this, it is really easy to add more opcodes. See:

  http://code.google.com/p/numexpr/issues/detail?id-28

  as an example.
- Added a couple of opcodes to VM: where_bbbb and cast_ib. The first allow to get boolean arrays out of the *where* function. The second allows to cast a boolean array into an integer one. Thanks to gdementen for his contribution.
- Fix negation of *int64* numbers. Closes #25.
- Using a *npy_intp* datatype (instead of plain *int*) so as to be able to manage arrays larger than 2 GB.

7.1.34 Changes from 1.3 to 1.3.1

- Due to an oversight, `uint32` types were not properly supported. That has been solved. Fixes #19.
- Function *abs* for computing the absolute value added. However, it does not strictly follow NumPy conventions. See `README.txt` or website docs for more info on this. Thanks to Pauli Virtanen for the patch. Fixes #20.

7.1.35 Changes from 1.2 to 1.3

- A new type called internally *float* has been implemented so as to be able to work natively with single-precision floating points. This prevents the silent upcast to *double* types that was taking place in previous versions, so allowing both an improved performance and an optimal usage of memory for the single-precision computations. However, the casting rules for floating point types slightly differs from those of NumPy. See:

  http://code.google.com/p/numexpr/wiki/Overview

  or the README.txt file for more info on this issue.

- Support for Python 2.6 added.

- When linking with the MKL, added a '-rpath' option to the link step so that the paths to MKL libraries are automatically included into the runtime library search path of the final package (i.e. the user won't need to update its LD_LIBRARY_PATH or LD_RUN_PATH environment variables anymore). Fixes #16.

7.1.36 Changes from 1.1.1 to 1.2

- Support for Intel's VML (Vector Math Library) added, normally included in Intel's MKL (Math Kernel Library). In addition, when the VML support is on, several processors can be used in parallel (see the new *set_vml_num_threads()* function). With that, the computations of transcendental functions can be accelerated quite a few. For example, typical speed-ups when using one single core for contiguous arrays are 3x with peaks of 7.5x (for the pow() function). When using 2 cores the speed-ups are around 4x and 14x respectively. Closes #9.

- Some new VML-related functions have been added:

  - set_vml_accuracy_mode(mode): Set the accuracy for VML operations.
  - set_vml_num_threads(nthreads): Suggests a maximum number of threads to be used in VML operations.
  - get_vml_version(): Get the VML/MKL library version.

  See the README.txt for more info about them.

- In order to easily allow the detection of the MKL, the setup.py has been updated to use the numpy.distutils. So, if you are already used to link NumPy/SciPy with MKL, then you will find that giving VML support to numexpr works almost the same.

- A new *print_versions()* function has been made available. This allows to quickly print the versions on which numexpr is based on. Very handy for issue reporting purposes.

- The *numexpr.numexpr* compiler function has been renamed to *numexpr.NumExpr* in order to avoid name collisions with the name of the package (!). This function is mainly for internal use, so you should not need to upgrade your existing numexpr scripts.

7.1.37 Changes from 1.1 to 1.1.1

- The case for multidimensional array operands is properly accelerated now. Added a new benchmark (based on a script provided by Andrew Collette, thanks!) for easily testing this case in the future. Closes #12.

- Added a fix to avoid the caches in numexpr to grow too much. The dictionary caches are kept now always with less than 256 entries. Closes #11.

- The VERSION file is correctly copied now (it was not present for the 1.1 tar file, I don't know exactly why). Closes #8.

7.1.38 Changes from 1.0 to 1.1

- Numexpr can work now in threaded environments. Fixes #2.
- The test suite can be run programmatically by using `numexpr.test()`.
- Support a more complete set of functions for expressions (including those that are not supported by MSVC 7.1 compiler, like the inverse hyperbolic or log1p and expm1 functions. The complete list now is:
  - **where(bool, number1, number2): number** Number1 if the bool condition is true, number2 otherwise.
  - **{sin,cos,tan}(float|complex): float|complex** Trigonometric sinus, cosinus or tangent.
  - **{arcsin,arccos,arctan}(float|complex): float|complex** Trigonometric inverse sinus, cosinus or tangent.
  - **arctan2(float1, float2): float** Trigonometric inverse tangent of float1/float2.
  - **{sinh,cosh,tanh}(float|complex): float|complex** Hyperbolic sinus, cosinus or tangent.
  - **{arcsinh,arccosh,arctanh}(float|complex): float|complex** Hyperbolic inverse sinus, cosinus or tangent.
  - **{log,log10,log1p}(float|complex): float|complex** Natural, base-10 and log(1+x) logarithms.
  - **{exp,expm1}(float|complex): float|complex** Exponential and exponential minus one.
  - **sqrt(float|complex): float|complex** Square root.
  - **{real,imag}(complex): float** Real or imaginary part of complex.
  - **complex(float, float): complex** Complex from real and imaginary parts.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index