hope Documentation

Release 0.6.1

Lukas Gamper

July 04, 2016

Contents

HOPE is a specialized method-at-a-time JIT compiler written in Python for translating Python source code into C++ and compiles this at runtime. In contrast to other existing JIT compliers, which are designed for general purpose, we have focused our development of the subset of the Python language that is most relevant for astrophysical calculations. By concentrating on this subset, **HOPE** is able to achieve the highest possible performance

By using **HOPE**, the user can benefit from being able to write common numerical code in Python and having the performance of compiled implementation. To enable the **HOPE** JIT compilation, the user needs to add a decorator to the function definition. The package does not require additional information, which ensures that **HOPE** is as non-intrusive as possible:

```
from hope import jit

@jit
def sum(x, y):
    return x + y
```

The **HOPE** package has been developed at ETH Zurich in the Software Lab of the Cosmology Research Group of the ETH Institute of Astronomy, and is now publicly available at GitHub.

Further information on the package can be found in our paper, on readthedocs.org and on our website.

CHAPTER 1

Installation

The package has been uploaded to PyPI and can be installed at the command line via pip:

```
$ pip install hope
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv hope
$ pip install hope
```

CHAPTER 2

Contents

2.1 Documentation

hope.**jit**(*fkt*)

Compiles a function to native code and return the optimized function. The new function has the performance of a compiled function written in C.

Parameters **fkt** (*function*) – function to compile to c

Returns function – optimized function

This function can either be used as decorator

```
@jit
def sum(x, y):
    return x + y
```

or as a normal function

```
def sum(x, y):
    return x + y
sum_opt = jit(sum)
```

hope.**serialize**(*obj*, *name*)

Write a pickled representation of obj to a file named `name` inside `hope.config.prefix`

Parameters

- **obj** (*mixed*) – arbitrary object to serialize
- **name** (*str*) – name of the object

hope.**unserialize**(*name*)

Read an object named `name` form `hope.config.prefix`. If the file does not exits `unserialize` returns `None`

Parameters **name** (*str*) – name of the object

Returns mixed – unserialized object

2.1.1 hope.config

hope.config.**cxxflags = [u'-Wall', u'-Wno-unused-variable', u'-std=c++11']**

List of c++ compiler flags. Normally hope does determing the right flags itself.

`hope.config.`**`hopeless`** **= False**
Disable hope. If hope.config.hopeless is True, hope.jit return the original function. Use this function for debug purpos

`hope.config.`**`keeptemp`** **= False**
Keep the intermediate c++ source and compiler output generated during compilation.

`hope.config.`**`optimize`** **= False**
Use ‘’‘sympy’‘’ to simplify expression and exptract common subexpression detection

`hope.config.`**`prefix`** **= u'.hope'**
Prefix of the folder hope saves all data in.

`hope.config.`**`rangecheck`** **= False**
Check if indeces are out of bounds

`hope.config.`**`verbose`** **= False**
Print a intermediate representation of each function during compilation.

2.2 Language Specification

This document specifies the Python subset supported by **HOPE**.

2.2.1 Native Types

The following native types are supported

- `bool`
- `int`
- `float`

2.2.2 NumPy Types

The following NumPy types are supported

- `bool_`
- `integer`, `signedinteger`, `byte`, `short`, `intc`, `intp`, `int0`, `int_`, `longlong`
- `int8`, `int16`, `int32`, `int64`
- `unsignedinteger`, `ubyte`, `ushort`, `uintc`, `uintp`, `uint0`, `uint_`, `ulonglong`
- `uint8`, `uint16`, `uint32`, `uint64`
- `single`, `float_`
- `float32`, `float64`
- `ndarray`

2.2.3 Conditional Expressions

- If
- If/Else

- If/ElseIf/Else

2.2.4 Loops

The `while` statement is supported as well as `for` loops but only with `range(stop)` or `range(start, stop)` resp. `xrange`:

```
for i in range(start, stop):
    foo()
```

2.2.5 Return Statement

A function needs to have a fixed return type. **HOPE** currently supports scalar and array data types as return values.

The following code will not compile as the type of the return value may change depending on the execution:

```
@hope.jit
def incompatible_return(arg):
    if arg > 10:
        return 1
    else:
        return 2.3 # ERROR: Inconsistent return type
```

2.2.6 Call functions

Call to pure Python functions are supported if the function

- is accessible form the global scope of the function
- has no decorators
- only uses the subset of Python supported by **HOPE**
- has no recursive or cyclic calls

Then the called function is also compiled to c++ and included in the shared object regardless where the function was defined originally.

2.2.7 Operators

Assignment

Assign	`b = a`

Unary operators

UAdd	`+a`
USub	`-a`

Binary operators

Add	`a + b`
Sub	`a - b`
Mult	`a * b`
Div	`a / b`
FloorDiv	`a // b`
Pow	`a ** b`
Mod	`a % b`
LShift	`a << b`
RShift	`a >> b`
BitOr	`a | b`
BitXor	`a ^ b`
BitAnd	`a & b`

Augmented assign statements

AugAdd	`a += b`
AugSub	`a -= b`
AugMult	`a *= b`
AugDiv	`a /= b`
AugFloorDiv	`a /= b`
AugPow	`a **= b`
AugMod	`a %= b`
AugLShift	`a <<= b`
AugRShift	`a <<= b`
AugBitOr	`a | b`
AugBitXor	`a ^ b`
AugBitAnd	`a & b`

Comparison Operators

Eq	`a == b`
NotEq	`a != b`
Lt	`a < b`
LtE	`a <= b`
Gt	`a > b`
GtE	`a >= b`

Bool Operators

&&	`a and b`
||	`a or b`

2.2.8 NumPy Array creation routines

`empty(shape[, dtype])`	Return a new array of given shape and type, without initializing entries.
`ones(shape[, dtype])`	Return a new array of given shape and type, filled with ones.
`zeros(shape[, dtype])`	Return a new array of given shape and type, filled with zeros.

2.2.9 NumPy Mathematical functions

Trigonometric functions

`sin(x)`	Trigonometric sine, element-wise.
`cos(x)`	Cosine elementwise.
`tan(x)`	Compute tangent element-wise.
`arcsin(x)`	Inverse sine, element-wise.
`arccos(x)`	Trigonometric inverse cosine, element-wise.
`arctan(x)`	Trigonometric inverse tangent, element-wise.

Hyperbolic functions

`sinh(x)`	Hyperbolic sine, element-wise.
`cosh(x)`	Hyperbolic cosine, element-wise.
`tanh(x)`	Compute hyperbolic tangent element-wise.

Exponents and logarithms

`exp(x)`	Calculate the exponential of all elements in the input array.

Miscellaneous

`sum(x)`	Return the sum of array elements.
`sqrt(x)`	Return the positive square-root of an array, element-wise.
`interp(x, xp, fp[, left, right])`	One-dimensional linear interpolation.
`ceil(x)`	Return the ceiling of the input, element-wise.
`floor(x)`	Return the floor of the input, element-wise.
`trunc(x)`	Return the truncated value of the input, element-wise.
`pi`	Returns the pi constant
`fabs`	Compute the absolute values element-wise
`sign`	Returns an element-wise indication of the sign of a number

2.2.10 Attributes of `numpy.ndarray`

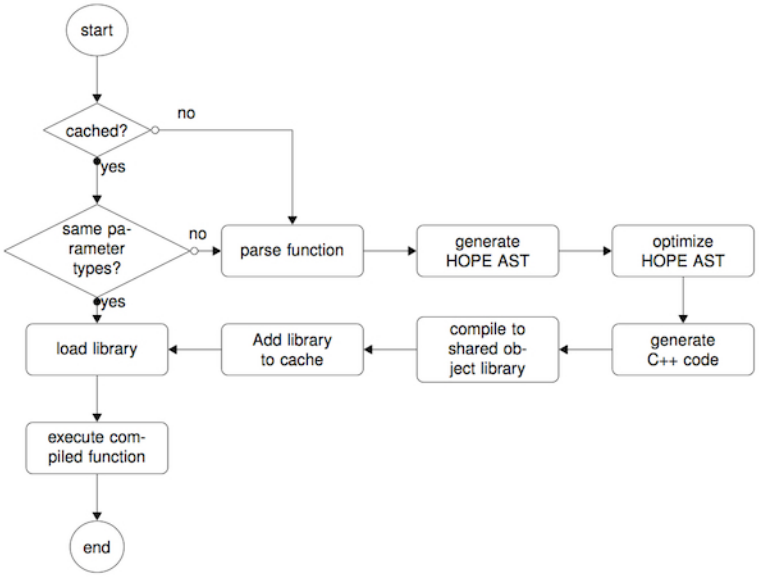
No attributes are supported at the moment

2.2.11 Others

- Added cast operators for np.bool_, np.int_, np.intc, np.int8, np.int16, np.int32, np.int64, np.uint8, np.uint16, np.uint32, np.uint64, np.float_, np.

2.3 Hope Architecture

The Just-in-time compiling process undergoes several steps. Those steps are explained in the following and detailed in an example and in a dedicated optimization section:



Start The Python interpreter loads a function or method previously decorated with the `@hope.jit` decorator.

Cache verification HOPE checks if a compiled version of the requested functions has previously been cached.

Parse function The first time the decorated function is called, the wrapper generates an abstract syntax tree (AST) by parsing the function definition using the Python built-in `ast` package.

Generate HOPE AST Using the visitor pattern, the Python AST is traversed and a corresponding **HOPE** specific AST is generated. During the traversal we use the Python built-in `inspect` package to infer the data types of the live objects such as parameters, variable and return values.

Numerical optimization HOPE traverses the new AST in order to identify numerical optimization possibilities and alters the tree accordingly.

Generate C++ code A C++ code is generated from the **HOPE** AST.

Compile code to shared object library The Python built-in `setuptools` package is then used to compile a shared object library from the generated code.

Add library to cache Using the extracted information from the function signature and a hash over the function body the compiled shared object library is cached for future calls.

Load library The shared object library is dynamically loaded into the runtime.

Execute compiled function A call to the function is directed to the function in the shared object library and executed with the passed parameters.

Subsequent function call HOPE analyzes the types of the passed arguments and queries the cache for a function matching the requested name and arguments.

2.3.1 Further reading

Example

Assume the following example:

poly.py

```
from hope import jit

def poly(x, y, a):
    x1 = x - a
    y[:] = x1 + x1 * x1

poly_hope = jit(poly)
```

Step-by-step evaluation

In the following we analyze the execution of the example.

call1.py

```
from poly import poly
import numpy as np

y = np.empty(1000, dtype=np.float32)
poly_hope(np.random.random(1000).astype(np.float32), y, 3.141)
```

Executing `python call1.py` will cause the following steps to happen:

When evaluating the statement `poly_hope = jit(poly)`

1. **HOPE** checks if a shared object of a compiled version of `poly` is available. Since we run it the first time no object is available, so **HOPE** returns a wrapper function that contains a reference to the original function.

When evaluating the statement `poly_hope(np.random.random(1000).astype(np.float32), y, 3.141)`

1. The wrapper function, which was returned by jit, is called
2. A Python AST of `poly` is generated:

```
        FunctionDef(
              name='poly'
            , args=arguments(args=[Name(id=x), Name(id=y), Name(id=a)])
            , body=[
                  Assign(
                      targets=[Name(id=x1)]
                    , value=BinOp(left=Name(id=x), op=Sub, right=Name(id=a))
                  )
                , Assign(
                      targets=[Subscript(value=Name(id=y)
                                         , slice=Slice(lower=None, upper=None, step=None))]
                    , value=BinOp(left=Name(id=x1)
                                  , op=Add
                                  , right=BinOp(left=Name(id=x1)
                                                , op=Mult
                                                , right=Name(id=x1)))
                  )
              ]
        )
```

3. The arguments passed to `poly` are analyzed:
    - `x`: `numpy.float32`, 1D

- y: numpy.float32, 1D
- a: numpy.float64, scalar (originally a has type float but this is equivalent to numpy.float64)

4. **HOPE** generates an identification for the arguments: f1f1d
5. **HOPE** generates a HOPE AST from the Python AST and the analyzed arguments:

```
Module(
      main=poly
    , functions=[
        FunctionDef(
              name='poly'
            , args=arguments(args=[
                  Variable(id=x, shape=(0, x_0), dtype=numpy.float32
                           , scope=signature, allocated=true)
                , Variable(id=y, shape=(0, y_0), dtype=numpy.float32
                           , scope=signature, allocated=true)
                , Variable(id=a, shape=(), dtype=numpy.float64
                           , scope=signature, allocated=true)
              ])
            , merged=[[(0, x_0), (0, y_0)]]
            , body=[
                  Block(body=[
                      Assign(
                          target=Variable(id=x1, shape=(0, x_0), dtype=numpy.float32
                                          , scope=block, allocated=false)
                        , value=BinOp(left=Variable(id=x, ...)
                                      , op=Sub, right=Variable(id=a, ...)
                                      , shape=(0, x_0), dtype=numpy.float32)
                      , Assign(
                          target=View(variable=Variable(id=y, ...)
                                      , extend=[0, y_0)
                                      , shape=(0, x_0), dtype=numpy.float32
                        , value=BinOp(
                              left=Variable(id=x, ...)
                            , op=Sub
                            , right=BinOp(left=Variable(id=x1, ...)
                                          , op=Mult, right=Variable(id=x1, ...)
                                          , shape=(0, x_0), dtype=numpy.float32)
                            , shape=(0, x_0), dtype=numpy.float32
                          )
                  ], shape=(0, x_0), dtype=numpy.float32)
              ]
        )
      ]
)
```

Differences between the Python AST and the HOPE AST:

- The **HOPE** AST is statically typed, each token has a scalar type (dtype) and for a start, stop for each dimension (shape) where shape=(0, x_0) means start=0, stop=x.shape[0]
- The function definition has a property *merged*. This list of lists identifies all segments (each dimension of a shape is called segment), which are equal. This is determined as follow:
  - the statement x1 = x - a implies that x1 has the same shape as x
  - the statement zz y[:] = x1 + x1 * x1 is only valid if x1 and y have the same shape.

  so x and y must have the same shape.

- The function body contains a `Block` token. This token is generated the following way:
  (a) Each statement in the body is wrapped into a `Block` token. Each `Block` token has the shape of the statement
  (b) All neighbor blocks with the same shape are merged
- Variables have a scope, which can either be:
  - `signature`: variables that are passed on call
  - `body`: variables, which occur in more than one Block
  - `block`: variables, which occur only in one Block token

6. **HOPE** traverses the new AST in order to identify numerical optimization possibilities optimization
7. generate C++11 code from the **HOPE** AST. The Block taken above is translated into the following C++ code:
   - the shape of `x` is stored in the `sx` array
   - the C pointer to the data of `x` is stored `cx`, `ca` is a double value containing the value of `a`

```
for (npy_intp i0 = 0; i0 < sx[0] - 0; ++i0) {
    auto cx1 = (cx[i0] - ca);
    cy[i0] = (cx1 + (cx1 * cx1));
}
```

   - The whole `Block` statement is turned into one loop over the shape of the block. This allows us to evaluate the operation element-wise, which improves cache locality.
   - For variables with `Block` scope there is no need to allocate a whole array, we only allocate a scalar value.
8. the C++ code is compiled into a shared object library
9. the shared object library is dynamically imported and the compiled function is evaluated.

call2.py

```
from poly import poly
import numpy as np

y = np.empty(1000, dtype=np.float32)
poly_hope(np.random.random(1000).astype(np.float32), y, 3.141)

y = np.empty(1000, dtype=np.float64)
poly_hope(np.random.random(1000).astype(np.float64), y, 42)
```

Executing `python call2.py` will cause the following steps to happen:

When evaluating the statement `poly_hope = jit(poly)`

1. checks if a shared object of a compiled version of `poly` is available. Since a shared object is available the shared object is dynamically loaded
2. a callback function for unknown signatures is registered in the module
3. the reference to the compiled `poly` function is returned

When evaluating the statement `poly_hope(np.random.random(1000).astype(np.float32), y, 3.141)`

1. the compiled `poly` function is called

When evaluating the statement `poly_hope(np.random.random(1000).astype(np.float64), y, 42)`

1. there is no compiled `poly` function for the passed argument types, so the registered callback is called
2. the arguments which are passed to `poly` are analysed:
   - `x`: `numpy.float64`, 1D
   - `y`: `numpy.float64`, 1D
   - `a`: `numpy.int64`, scalar (originally `a` has type `int` but this is equivalent to `numpy.int64`)
3. The code is regenerated as described above, but this time with two different function signatures. Once for
   - `x`: `numpy.float32`, 1D
   - `y`: `numpy.float32`, 1D
   - `a`: `numpy.float64`, scalar

   and once for
   - `x`: `numpy.float64`, 1D
   - `y`: `numpy.float64`, 1D
   - `a`: `numpy.int64`, scalar (originally `a` has type `int` but this is equivalent to `numpy.int64`)
4. The new shared object library is dynamically imported and evaluated

Optimization

After the **HOPE** specific AST has been created the package performs a static recursive analysis of the expressions to introduce numerical optimization. The supported possibilities are divided into three groups:

Simplification of expressions

To simplify expression we have used the `SymPy` library. SymPy is a Python library for symbolic mathematics and has been entirely written in Python. To apply the optimization, the AST expression is translated into `SymPy` syntax AST and passed to the `simplify` function. The function applies various different heuristics to reduce the complexity of the passed expression. The simplification is not exactly defined and varies depending on the input.

For instance, one example of simplification is that $sin(x)^2 + cos(x)^2$ will be simplified to 1.

Factorizing out subexpressions

Furthermore the `SymPy` library is used to factorize out recurring subexpression (common subexpression elimination) using the previously created `SymPy` AST and `SymPy`'s `cse` function.

Replacing the pow function for integer exponents

From C++11 on, the `pow` function in the C standard library is not overloaded for integer exponents. The internal implementation of the computation of a base to the power of a double exponent is typically done using a series expansion, though this may vary depending on the compiler and hardware architecture. Generally this is efficient for double exponents but not necessarily for integer exponents.

HOPE therefore tries to identify power expressions with integer exponents and factorizes the expression into several multiplications e.g. $y = x^5$ will be decomposed into $x_2 = x^2$ and $y = x_2 \times x_2 \times x$. This reduces the computational costs and increases the performance of the execution.

2.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.4.1 Types of Contributions

Report Bugs

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Implement Features

Write Documentation

HOPE could always use more documentation, whether as part of the official **HOPE** docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.4.2 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, and 3.3. make sure that the tests pass for all supported Python versions.

2.5 Credits

2.5.1 Development Lead

- Lukas Gamper <gamperl@gmail.com>
- Joel Akeret <jakeret@phys.ethz.ch>

2.5.2 Contributors

We would like to thank several people for helping to test this package before release.

- Alexandre Refregier
- Adam Amara
- Claudio Bruderer
- Chihway Chang
- Sebastian Gaebel
- Joerg Herbel

2.5.3 Citations

As you use **HOPE** for your exciting discoveries, please cite the paper that describes the package:

Akeret, J., Gamper, L., Amara, A. and Refregier, A., Astronomy and Computing (2015)

2.5.4 Feedback

If you have any suggestions or questions about **HOPE** feel free to email me at hope@phys.ethz.ch.

If you encounter any errors or problems with **HOPE**, please let me know!

2.6 History

2.6.1 0.6.1 (2016-07-04)

- fixing bug when accessing class members for operations

2.6.2 0.6.0 (2016-04-19)

- Fixed bug in 2d array slicing
- Array slicing with negative index
- Fixed name clash bug with object attributes
- Replaced assignment with reference to object attributes

2.6.3 0.5.0 (2016-01-20)

- Fixed memory leak when creating array in jitted fkt
- Fixed incorrect bound handling in numpy.interp

2.6.4 0.4.0 (2015-02-04)

- Increased compilation speed for large functions
- Support for variable allocation within if-else
- Added support for numpy.sign
- Updated Cython implementation in benchmarks
- Fixed array allocation problem under OSX Yosemite (thx iankronquist)

2.6.5 0.3.1 (2014-10-24)

- Better support for Python 3.3 and 3.4
- Proper integration in Travis CI
- Improved support for Linux systems (*accepting x86_64-linux-gnu-gcc*)
- Avoiding warning on Linux by removing *Wstrict-prototypes* arg
- Supporting gcc proxied clang (OS X)
- Added set of examples

2.6.6 0.3.0 (2014-10-16)

- Language: scalar return values
- Shared libraries are written to hope.config.prefix
- Function call can have return values
- Fixed function calls to function with no arguments
- Make sure code is recompiled if the python code has changed
- Added config.optimize to simplify expression using sympy and replace pow
- Speed improvements for hope
- Added support for object properties
- Added support for object methods
- Addes support for True and False
- Addes support for While
- Addes support for numpy.sum
- Addes support for numpy.pi
- Added support for numpy.floor, numpy.ceil, numpy.trunc, numpy.fabs, numpy.log
- improved error messages

- Added config.rangecheck flag
- Support xrange in for loop
- Added cast operators for np.bool_, np.int_, np.intc, np.int8, np.int16, np.int32, np.int64, np.uint8, np.uint16, np.uint32, np.uint64, np.float_, np.float32, np.float64,
- Added bool operators
- Added the following operators:

FloorDiv	`a // b`
Mod	`a % b`
LShift	`a << b`
RShift	`a >> b`
BitOr	`a | b`
BitXor	`a ^ b`
BitAnd	`a & b`
AugFloorDiv	`a //= b`
AugPow	`a **= b`
AugMod	`a %= b`
AugLShift	`a <<= b`
AugRShift	`a <<= b`
AugBitOr	`a | b`
AugBitXor	`a ^ b`
AugBitAnd	`a & b`

2.6.7 0.2.0 (2014-03-05)

- First release on private PyPI.

2.6.8 0.1.0 (2014-02-27)

- Initial creation.

CHAPTER 3

Benchmarks

All the benchmarks have been made available online as IPython notebooks (also on GitHub). If you would like to run the benchmarks on your machine, you can download the notebooks in the nbviewer.

Note: Make sure to execute the *native_cpp_gen* the first time in order to generate the native C++ code and some Python utility modules.

h

C

H

J

K

O

P

R

S

U

V