EBook Gratis

APRENDIZAJE cython

Free unaffiliated eBook created from
Stack Overflow contributors.

#cython

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cython

It is an unofficial and free cython ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cython.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con cython

Observaciones

¿Qué es Cython?

El lenguaje de programación Cython enriquece la escritura estática tipo Python by C, la capacidad de llamar directamente a las funciones C y muchas otras características. Esto permite alcanzar un rendimiento de nivel C sin dejar de utilizar una sintaxis similar a Python.

¿Como funciona?

El código de Cython se compila utilizando el compilador de fuente a fuente de cython para crear código C o C ++, que a su vez se puede compilar utilizando un compilador de C. Esto permite crear extensiones que se pueden importar desde Python o ejecutables.

La principal ganancia de rendimiento que Cython puede alcanzar en contraste con Python puro se deriva de pasar por alto la API de CPython. Por ejemplo, al agregar dos enteros, Python realiza una verificación de tipo para cada variable, encuentra una función de adición que satisface los tipos encontrados y llama a esa función. En el código C generado por Cython, los tipos ya se conocen y solo se realiza una llamada a la función. Por lo tanto, Cython brilla especialmente para problemas matemáticos en los que los tipos son claros.

¿Cómo lo uso para acelerar mi código?

Un caso de uso común, cuando se intenta acelerar un programa usando Cython, es perfilar el código y mover las partes computacionalmente caras a los módulos compilados de Cython. Esto permite conservar la sintaxis de Python para la mayor parte del código y aplicar la aceleración donde más se necesita.

Examples

Instalando Cython

Para usar Cython se necesitan dos cosas. El propio paquete de Cython, que contiene el `cython` fuente a fuente y las interfaces de Cython a varias bibliotecas de C y Python (por ejemplo, numpy). Para compilar el código C generado por el compilador de `cython` , se necesita un compilador de C.

Paso 1: Instalando Cython

Sistema agnóstico

Cython se puede instalar con varios sistemas de gestión de paquetes independientes del sistema. Éstos incluyen:

1. PyPI a través de pip o easy_install:

```
$ pip install cython
$ easy_install cython
```

2. anaconda utilizando conda:

```
$ conda install cython
```

3. Piensa en dosel usando el gestor de paquetes enpkg:

```
$ enpkg cython
```

También el código fuente se puede descargar desde github e instalarse manualmente usando:

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

Ubuntu, Debian

Para Ubuntu los paquetes `cython` y `cython3` están disponibles. Tenga en cuenta que estos proporcionan una versión más antigua que las opciones de instalación mencionadas anteriormente.

```
$ apt-get install cython cython3
```

Windows

Para Windows, un archivo .whl que se puede instalar utilizando pip es proporcionado por un tercero. Los detalles sobre la instalación de un archivo .whl en Windows se pueden encontrar aquí .

Paso 2: Instalar un compilador de C

Para compilar los archivos C generados por Cython, se necesita un compilador para C y C ++. Se recomienda el compilador gcc y se puede instalar de la siguiente manera.

Ubuntu, Debian

El paquete `build-essential` contiene todo lo que se necesita. Se puede instalar desde los repositorios utilizando:

```
$ sudo apt-get install build-essential
```

MAC

Las herramientas de desarrollo de XCode contienen un compilador similar a gcc.

Windows

MinGW (Minimalist GNU para Windows) contiene una versión de Windows de gcc. También se puede utilizar el compilador de Visual Studio.

Hola Mundo

Un archivo Pyth de Cython debe traducirse al código C (*citonizado*) y compilarse antes de que se pueda usar desde Python. Un enfoque común es crear un módulo de extensión que luego se importa en un programa Python.

Código

Para este ejemplo creamos tres archivos:

- `hello.pyx` contiene el código de Cython.
- `test.py` es un script de Python que usa la extensión hello.
- `setup.py` se utiliza para compilar el código de Cython.

hola.pyx

```
from libc.math cimport pow

cdef double square_and_add (double x):
    """Compute x^2 + x as double.

    This is a cdef function that can be called from within
    a Cython program, but not from Python.
    """
    return pow(x, 2.0) + x

cpdef print_result (double x):
    """This is a cpdef function that can be called from Python."""
    print("({} ^ 2) + {} = {}".format(x, x, square_and_add(x)))
```

test.py

```
# Import the extension module hello.
import hello

# Call the print_result method
hello.print_result(23.0)
```

setup.py

```
from distutils.core import Extension, setup
from Cython.Build import cythonize

# define an extension that will be cythonized and compiled
ext = Extension(name="hello", sources=["hello.pyx"])
setup(ext_modules=cythonize(ext))
```

Compilando

Esto se puede hacer usando `cython hello.pyx` para traducir el código a C y luego compilarlo usando `gcc` . Una forma más fácil es dejar que distutils maneje esto:

```
$ ls
hello.pyx  setup.py  test.py
$ python setup.py build_ext --inplace
$ ls
build  hello.c  hello.cpython-34m.so  hello.pyx  setup.py  test.py
```

El archivo de objeto compartido (.so) se puede importar y usar desde Python, por lo que ahora podemos ejecutar el `test.py` :

```
$ python test.py
(23.0 ^ 2) + 23.0 = 552.0
```

Lea Empezando con cython en línea: https://riptutorial.com/es/cython/topic/2925/empezando-con-cython

Capítulo 2: Agrupación de cython

Examples

Agrupar un programa Cython usando pyinstaller

Comience desde un programa Cython con un punto de entrada:

```
def do_stuff():
    cdef int a,b,c
    a = 1
    b = 2
    c = 3
    print("Hello World!")
    print([a,b,c])
    input("Press Enter to continue.")
```

Crea un archivo `setup.py` en la misma carpeta:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    name = "Hello World",
    ext_modules = cythonize('program.pyx'),
)
```

Ejecutarlo con `python setup.py build_ext --inplace` producirá una biblioteca `.pyd` en una subcarpeta.

Después de eso, cree una secuencia de comandos de Python de vainilla utilizando la biblioteca (por ejemplo, `main.py`) y coloque el archivo `.pyd` al lado:

```
import program
program.do_stuff()
```

Use PyInstaller para agruparlo en `pyinstaller --onefile "main.py"` . Esto creará una subcarpeta que contiene el ejecutable de un tamaño de 4 MB + que contiene la biblioteca más el tiempo de ejecución de python.

Generación automatizada (Windows)

Para la automatización del procedimiento anterior en Windows, use un `.bat` de los contenidos similares:

```
del "main.exe"
python setup.py build_ext --inplace
del "*.c"
rmdir /s /q ".\build"
pyinstaller --onefile "main.py"
```

```
copy /y ".\dist\main.exe" ".\main.exe"
rmdir /s /q ".\dist"
rmdir /s /q ".\build"
del "*.spec"
del "*.pyd"
```

Agregando Numpy al paquete

Para agregar Numpy al paquete, modifique el `setup.py` con la palabra clave `include_dirs` e importe el número necesario en la secuencia de comandos de Python para notificar a Pyinstaller.

`program.pyx` :

```
import numpy as np
cimport numpy as np


def do_stuff():
    print("Hello World!")
    cdef int n
    n = 2
    r = np.random.randint(1,5)
    print("A random number: "+str(r))
    print("A random number multiplied by 2 (made by cdef):"+str(r*n))
    input("Press Enter to continue.")
```

`setup.py` :

```
from distutils.core import setup, Extension
from Cython.Build import cythonize
import numpy

setup(
    ext_modules=cythonize("hello.pyx"),
    include_dirs=[numpy.get_include()]
)
```

`main.py` :

```
import program
import numpy
program.do_stuff()
```

Lea Agrupación de cython en línea: https://riptutorial.com/es/cython/topic/6386/agrupacion-de-cython

Capítulo 3: Envolviendo C ++

Examples

Envolviendo un DLL: C ++ a Cython a Python

Esto demuestra un ejemplo no trivial de envolver una dll de C ++ con Cython. Cubrirá los siguientes pasos principales:

- Crea un ejemplo de DLL con C ++ usando Visual Studio.
- Envuelva la DLL con Cython para que pueda ser llamada en Python.

Se supone que tienes Cython instalado y puedes importarlo con éxito en Python.

Para el paso de DLL, también se asume que está familiarizado con la creación de una DLL en Visual Studio.

El ejemplo completo incluye la creación de los siguientes archivos:

1. `complexFunLib.h` : archivo de encabezado para la fuente DLL de C ++
2. `complexFunLib.cpp` : archivo CPP para la fuente DLL de C ++
3. `ccomplexFunLib.pxd` : archivo "cabecera" de Cython
4. `complexFunLib.pyx` : archivo "contenedor" de Cython
5. `setup.py` : archivo de configuración de Python para crear `complexFunLib.pyd` con Cython
6. `run.py` : Ejemplo de archivo Python que importa la DLL compilada y envuelta en Cython

C ++ DLL Fuente: `complexFunLib.h` y `complexFunLib.cpp`

Salta esto si ya tienes una DLL y un archivo fuente de cabecera. Primero, creamos la fuente de C ++ a partir de la cual se compilará la DLL utilizando Visual Studio. En este caso, queremos hacer cálculos de matriz rápidos con la función exponencial compleja. Las dos funciones siguientes realizan el cálculo `k*exp(ee)` en las matrices `k` y `ee` , donde los resultados se almacenan en `res` . Hay dos funciones para acomodar tanto la precisión simple como la doble. Tenga en cuenta que estas funciones de ejemplo utilizan OpenMP, así que asegúrese de que OpenMP esté habilitado en las opciones de Visual Studio para el proyecto.

Archivo H

```
// Avoids C++ name mangling with extern "C"
#define EXTERN_DLL_EXPORT extern "C" __declspec(dllexport)
#include <complex>
#include <stdlib.h>

// Handles 64 bit complex numbers, i.e. two 32 bit (4 byte) floating point numbers
EXTERN_DLL_EXPORT void mp_mlt_exp_c4(std::complex<float>* k,
                                     std::complex<float>* ee,
                                     int sz,
                                     std::complex<float>* res,
```

```
                                    int threads);

// Handles 128 bit complex numbers, i.e. two 64 bit (8 byte) floating point numbers
EXTERN_DLL_EXPORT void mp_mlt_exp_c8(std::complex<double>* k,
std::complex<double>* ee,
                                    int sz,
                                    std::complex<double>* res,
                                    int threads);
```

Archivo CPP

```
#include "stdafx.h"
#include <stdio.h>
#include <omp.h>
#include "complexFunLib.h"

void mp_mlt_exp_c4(std::complex<float>* k,
                   std::complex<float>* ee,
                   int sz,
                   std::complex<float>* res,
                   int threads)
{
    // Use Open MP parallel directive for multiprocessing
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp for
        for (int i = 0; i < sz; i++) res[i] = k[i] * exp(ee[i]);
    }
}

void mp_mlt_exp_c8(std::complex<double>* k,
                   std::complex<double>* ee,
                   int sz, std::complex<double>* res,
                   int threads)
{
    // Use Open MP parallel directive for multiprocessing
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp for
        for (int i = 0; i < sz; i++) res[i] = k[i] * exp(ee[i]);
    }
}
```

Fuente de Cython: `ccomplexFunLib.pxd` y `complexFunLib.pyx`

A continuación, creamos los archivos de origen de Cython necesarios para envolver la DLL de C ++. En este paso, hacemos las siguientes suposiciones:

- Has instalado Cython
- Usted posee un archivo DLL de trabajo, por ejemplo, el descrito anteriormente

El objetivo final es crear el uso de estos archivos de origen de Cython junto con la DLL original para compilar un archivo `.pyd` que puede importarse como un módulo de Python y exponer las funciones escritas en C ++.

Archivo PXD

Este archivo corresponde al archivo de cabecera de C ++. En la mayoría de los casos, puede copiar y pegar el encabezado en este archivo con cambios menores específicos de Cython. En este caso, se utilizaron los tipos específicos de complejo de Cython. Tenga en cuenta la adición de `c` al principio de `ccomplexFunLib.pxd` . Esto no es necesario, pero hemos encontrado que tal convención de nombres ayuda a mantener la organización.

```
cdef extern from "complexFunLib.h":
    void mp_mlt_exp_c4(float complex* k, float complex* ee, int sz,
                       float complex* res, int threads);
    void mp_mlt_exp_c8(double complex* k, double complex* ee, int sz,
                       double complex* res, int threads);
```

Archivo PYX

Este archivo corresponde al archivo fuente de C ++ `cpp` . En este ejemplo, pasaremos punteros a los objetos ndarray `ndarray` a las funciones DLL de importación. También es posible utilizar el objeto incorporado en Cython `memoryview` para arrays, pero su rendimiento puede no ser tan bueno como los objetos `ndarray` (sin embargo, la sintaxis es significativamente más limpia).

```
cimport ccomplexFunLib  # Import the pxd "header"
# Note for Numpy imports, the C import most come AFTER the Python import
import numpy as np  # Import the Python Numpy
cimport numpy as np  # Import the C Numpy

# Import some functionality from Python and the C stdlib
from cpython.pycapsule cimport *

# Python wrapper functions.
# Note that types can be delcared in the signature

def mp_exp_c4(np.ndarray[np.complex64_t, ndim=1] k,
              np.ndarray[np.complex64_t, ndim=1] ee,
              int sz,
              np.ndarray[np.complex64_t, ndim=1] res,
              int threads):
    '''
    TODO: Python docstring
    '''
    # Call the imported DLL functions on the parameters.
    # Notice that we are passing a pointer to the first element in each array
    ccomplexFunLib.mp_mlt_exp_c4(&k[0], &ee[0], sz, &res[0], threads)

def mp_exp_c8(np.ndarray[np.complex128_t, ndim=1] k,
              np.ndarray[np.complex128_t, ndim=1] ee,
              int sz,
              np.ndarray[np.complex128_t, ndim=1] res,
              int threads):
    '''
    TODO: Python docstring
    '''
    ccomplexFunLib.mp_mlt_exp_c8(&k[0], &ee[0], sz, &res[0], threads)
```

Fuente de Python: `setup.py` y `run.py`

setup.py

Este archivo es un archivo de Python que ejecuta la compilación de Cython. Su propósito es generar el archivo `.pyd` compilado que luego puede ser importado por los módulos de Python. En este ejemplo, hemos mantenido todos los archivos necesarios (es decir `complexFunLib.h` , `complexFunLib.dll` , `ccomplexFunLib.pxd` y `complexFunLib.pyx`) en el mismo directorio que `setup.py` .

Una vez que se crea este archivo, debe ejecutarse desde la línea de comandos con los parámetros: `build_ext --inplace`

Una vez que se ejecuta este archivo, debe producir un archivo `.pyd` sin generar ningún error. Tenga en cuenta que en algunos casos, si hay un error, se puede crear `.pyd` pero no es válido. Asegúrese de que no se hayan generado errores en la ejecución de `setup.py` antes de usar el `.pyd` generado.

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy as np

ext_modules = [
    Extension('complexFunLib',
              ['complexFunLib.pyx'],
              # Note here that the C++ language was specified
              # The default language is C
              language="c++",
              libraries=['complexFunLib'],
              library_dirs=['.'])
    ]

setup(
    name = 'complexFunLib',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules,
    include_dirs=[np.get_include()]  # This gets all the required Numpy core files
)
```

run.py

Ahora `complexFunLib` puede ser importado directamente a un módulo de Python y las funciones DLL envueltas son llamadas.

```
import complexFunLib
import numpy as np

# Create arrays of non-trivial complex numbers to be exponentiated,
# i.e. res = k*exp(ee)
k = np.ones(int(2.5e5), dtype='complex64')*1.1234 + np.complex64(1.1234j)
ee = np.ones(int(2.5e5), dtype='complex64')*1.1234 + np.complex64(1.1234j)
sz = k.size  # Get size integer
res = np.zeros(int(2.5e5), dtype='complex64')  # Create array for results

# Call function
complexFunLib.mp_exp_c4(k, ee, sz, res, 8)

# Print results
print(res)
```

Lea Envolviendo C ++ en línea: https://riptutorial.com/es/cython/topic/3525/envolviendo-c-plusplus

Capítulo 4: Envolviendo Código C

Examples

Usando funciones de una biblioteca C personalizada

Tenemos una biblioteca de C llamada `my_random` que produce números aleatorios a partir de una distribución personalizada. Proporciona dos funciones que queremos usar: `set_seed(long seed)` y `rand()` (y muchas más que no necesitamos). Para usarlos en Cython necesitamos

1. definir una interfaz en el archivo .pxd y
2. llamar a la función en el archivo .pyx.

Código

test_extern.pxd

```
# extern blocks define interfaces for Cython to C code
cdef extern from "my_random.h":
    double rand()
    void c_set_seed "set_seed" (long seed) # rename C version of set_seed to c_set_seed to
avoid naming conflict
```

test_extern.pyx

```
def set_seed (long seed):
    """Pass the seed on to the c version of set_seed in my_random."""
    c_set_seed(seed)

cpdef get_successes (int x, double threshold):
    """Create a list with x results of rand <= threshold

    Use the custom rand function from my_random.
    """
    cdef:
        list successes = []
        int i
    for i in range(x):
        if rand() <= threshold:
            successes.append(True)
        else:
            successes.append(False)
    return successes
```

Lea Envolviendo Código C en línea: https://riptutorial.com/es/cython/topic/3626/envolviendo-codigo-c

Creditos

S. No	Capítulos	Contributors
1	Empezando con cython	Community, J.J. Hakala, Keith L, m00am
2	Agrupación de cython	Andrii Magalich
3	Envolviendo C ++	J.J. Hakala, Keith L, Kevin Pasquarella
4	Envolviendo Código C	m00am