Ejecución Nativa

Antonio Espín Herranz

Contenidos

- Numba
- Cython
- Desarrollo de módulos de extensión en C/C++

Numba

- Es una librería dedicada a la aceleración de ejecución del código Python.

- Compila el código o la función de Python antes de ejecutarla.
  - Es un tipo de compilador **just-in-time**.
  - Funciona con compiladores de tipo LLVM.

- A la hora de instalar puede dar problemas por la cantidad de dependencias que tiene:
  - La instalación mejor desde conda.
  - **conda install numba**

Cuando utilizar numba

- Debido a la naturaleza dinámica de Python cuando tenemos bucles anidados podemos obtener un mejor rendimiento con numba.

- Si utilizamos el compilador **just-in-time** de numba podemos obtener funciones altamente eficientes.

Comparativa

- OJO con los bucles anidados

- Generar un array de con numpy de 1000, 1000 e implementar 3 funciones:
  - Suma_numpy: suma el array con np.sum()
  - Suma_Python: suma con dos bucles anidados
  - Suma_numba: idem del anterior pero utilizamos el decorador:
    - import numba

    - **@numba.jit**
    - def suma_numba():
      - ….

En la primera ejecución compila la función
Compilador al vuelo

Comparativa II

- Desde la consola de ipython cargar el módulo con las 3 funciones.
- Y ejecutar: %timeit función_numba(…)

```
In [3]: %timeit suma_numba(np.random.rand(1000,1000))
11.9 ms ± 78.5 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [4]: %timeit suma_numpy(np.random.rand(1000,1000))
11.7 ms ± 35.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [5]: %timeit suma_python(np.random.rand(1000,1000))
224 ms ± 29.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

@numba.jit

- La primera vez que llamamos a la función tiene que detectar cuales son los tipos que están entrando y realizar las optimizaciones …

- Se pueden especificar los tipos dentro del decorador para ahorrar tiempo a numba en detectar cuales son los tipos que le estamos pasando.

- En el decorador se especifica que devuelve un float (en numba es f8) y recibe un array de float de 2 dimensiones (en numba f8[:, :])

```
@numba.jit('f8(f8[:, :])')
def suma_numba(arr):
    ….
```

@numba.jit

- Si numba espera un tipo float como parámetro y le pasamos un array de enteros → da error.

- En la llamada a la función podemos convertir el tipo del array para que lo reciba como float:
  - suma_numba(arr.astype(float))

Tipos de numba

Type name(s)	Shorthand	Comments
boolean	b1	represented as a byte
uint8, byte	u1	8-bit unsigned byte
uint16	u2	16-bit unsigned integer
uint32	u4	32-bit unsigned integer
uint64	u8	64-bit unsigned integer
int8, char	i1	8-bit signed byte
int16	i2	16-bit signed integer
int32	i4	32-bit signed integer
int64	i8	64-bit signed integer
intc	–	C int-sized integer
uintc	–	C int-sized unsigned integer
intp	–	pointer-sized integer
uintp	–	pointer-sized unsigned integer
float32	f4	single-precision floating-point number
float64, double	f8	double-precision floating-point number
complex64	c8	single-precision complex number
complex128	c16	double-precision complex number

@njit / @numba.jit(nopython=True)

- Para compilar en modo no Python, modo estricto.
- Con esto avisamos a numba que si detecta que alguna de las estructuras internas de los bucles no se puede optimizar emitirá un error.
  - En caso contrario de no avisarlo numba va a generar un código más complejo y este no estará tan optimizado.

- El código candidato a optimizar con numba son bucles anidados de varios niveles con llamadas a funciones de la librería.

- Se aconseja probar siempre primero este modo estricto. Es como podemos ganar la máxima optimización en numba.

- Probar a multiplicar 2 matrices de 10000 x 10000 elementos.

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.
- https://cython.readthedocs.io/en/latest/

Cython

- 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.

- Para acelerar el código Python se migra el código pesado y más lento a Cython, esto permite conservar la sintaxis de Python para la mayor parte del código y aplicar la aceleración donde más se necesita.

Ejemplos

- **Algoritmos de C**
  - https://github.com/fragglet/c-algorithms

¿Qué necesitamos para integrar C y Python?

- Un fichero **fuente en C** y el fichero de **cabecera .H**

- Un archivo **pxd**: es una interface para **cython**
  - Muy parecido al fichero .H de encabezado
  - No es necesario proporcionar todas las declaraciones como se indicó anteriormente, solo aquellas que usa en su código o en otras declaraciones.

- Un archivo **pyx:** es la clase envoltorio,
  - Sustituye el constructor **__init__** por **__cinit__**

- Un archivo **setup.py** para compilar el código.

- Para compilar: **python setup.py build_ext -i**

queue.h

- **Partimos de un fichero de C**

```
typedef struct _Queue Queue;
typedef void *QueueValue;

Queue *queue_new(void);
void queue_free(Queue *queue);

int queue_push_head(Queue *queue, QueueValue data);
QueueValue queue_pop_head(Queue *queue);
QueueValue queue_peek_head(Queue *queue);

int queue_push_tail(Queue *queue, QueueValue data);
QueueValue queue_pop_tail(Queue *queue);
QueueValue queue_peek_tail(Queue *queue);

int queue_is_empty(Queue *queue);
```

cqueue.pxd

cdef extern from "c-algorithms/src/queue.h": → **Indicar el path al fichero .h**

```
    ctypedef struct Queue:
        pass
    ctypedef void* QueueValue

    Queue* queue_new()
    void queue_free(Queue* queue)

    int queue_push_head(Queue* queue, QueueValue data)
    QueueValue  queue_pop_head(Queue* queue)
    QueueValue queue_peek_head(Queue* queue)

    int queue_push_tail(Queue* queue, QueueValue data)
    QueueValue queue_pop_tail(Queue* queue)
    QueueValue queue_peek_tail(Queue* queue)

    bint queue_is_empty(Queue* queue)
```

Clase envoltorio: queue.pyx

```
# distutils: sources = algoritmos/queue.c   → Path al fuente en C
# distutils: include_dirs = algoritmos/   → Carpeta de los algoritmos C

cimport cqueue

cdef class Queue:
    cdef cqueue.Queue* _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.queue_new()
        if self._c_queue is NULL:
            raise MemoryError()

    def __dealloc__(self):
        if self._c_queue is not NULL:
            cqueue.queue_free(self._c_queue)
```

setup.py

- Archivo setup para compilar el código:

```
from setuptools import Extension, setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize([Extension("queue", ["queue.pyx"])])
)
```

Para compilar: python setup.py build_ext -i

Para comprobar: python -c 'import queue; Q = queue.Queue()'

prueba_queue.py

- import queue
- Q = queue.Queue()
- print(Q)

Enlaces

- Numba
  - https://www.youtube.com/watch?v=cuiXCZz93tU&list=PLGBbVX_WvN7as_DnOGcpkSsUyXB1G_wqb&index=15
  - https://www.youtube.com/watch?v=cS7muqv8JzI&list=PLGBbVX_WvN7as_DnOGcpkSsUyXB1G_wqb&index=16

- https://realpython.com/build-python-c-extension-module/
- https://docs.microsoft.com/es-es/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2019

- Interfaces con otros lenguajes
  - https://scipy-cookbook.readthedocs.io/items/idx_interfacing_with_other_languages.html

- https://riptutorial.com/es/cython