monte_carlo

September 24, 2023

1 Parallelizzazione con Numba

[ ]:
```
import numpy as np
from numba import njit, prange
import math
```

[ ]:
```
@njit(parallel = True)
def replicate(sample_size:int,number_of_simulation:int,a:int,b:
 ↪int,simulation_results:np.ndarray[np.float32])->np.float32:
    for i in prange(number_of_simulation):
        np.random.seed(111+i)
        simulated_data = np.random.uniform(low=a,high=b,size=(sample_size,))
        simulation_results[i] = simulated_data.mean()
    return simulation_results.mean()
```

In questa piccola modifica abbiamo detto al compilatore che questa è una funzione che deve essere parallelizzata sulla CPU e se non riesce allora deve restituire un errore. Tutto questo è scritto nell'annotazione `@njit` che significa *non python just in time compiler*. La funzione `prange` di `numba` è l'equivalente di `range`, ma in parallelo.

I cicli `for` così costruiti, ovvero con operazioni indipendenti, si possono parallelizzare senza problemi.

[ ]:
```
sample_size = 1000
number_of_simulation = 10000
a = 1
b = 2
m_U = (a + b) / 2
sd_U = math.sqrt((b - a) ** 2 / 12)
sd_m_U = m_U / math.sqrt(sd_U)

simulation_results = np.empty(shape=(number_of_simulation,),dtype=np.float32)
media_simulata =␣
 ↪replicate(sample_size,number_of_simulation,a,b,simulation_results)

print(
    f"Media simulata: {media_simulata}\n" +
    f"Media reale: {m_U}"
```

```
)
```

```
Media simulata: 1.5000369397997857
Media reale: 1.5
```

Adesso vediamo se la funzione parallela restituisce lo stesso valore della funzione non parallelizzata. Numba conserva la versione non parallelizzata nell'attributo `.py_func`.

Per il testing utiliziamo il modulo `testing` di `numpy`. La funzione `testing.assert_almost_equal` restituisce un' eccezzione se i due valori in virgola mobile non sono uguali entro una certa tolleranza specificata.

[ ]:
```
from numpy import testing

testing.assert_almost_equal(
    replicate(sample_size,number_of_simulation,a,b,simulation_results),
    replicate.py_func(sample_size,number_of_simulation,a,b,simulation_results),
    decimal=5
    )
```

Scegliendo una tolleranza alla quinta cifra decimale, i due valori sono gli stessi.

Come ultimo test, vediamo che speed up abbiamo ottenuto parallelizzando questa simulazione.

Utilizziamo il magic command `%timeit`.

[ ]:
```
%timeit replicate(sample_size,number_of_simulation,a,b,simulation_results)
%timeit replicate.
  ↪py_func(sample_size,number_of_simulation,a,b,simulation_results)
```

```
8.66 ms ± 879 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
120 ms ± 599 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Abbiamo ottenuto uno speed up di più di 10 volte!

Nel caso in cui si volesse parallelizzare sulla GPU si devono prendere accortezze differenti sia per le operazioni possibili (generare numeri casuali sulla GPU non è possibile in maniera semplice), e ai trasferimenti dei dati (la GPU ha la sua memoria e non una la RAM del processore, quindi i dati che utilizza vanno spostati nella VRAM e riposrtati in RAM).

2 Reference

1. Numba documentation