NUMBA

Principe

Numba est une idée surprenante permettant d'accélérer le code Python assez facilement.

Mais comment procède Numba ? Lorsqu'une fonction Python est précédée du mot clef `@jit`, Numba traduit alors cette fonction en langage C (pour faire simple) puis la compile et la réintègre dans le programme Python.

Cette optimisation a lieu lors du premier appel de la fonction dans le script Python. Les types des arguments transmis servent alors à déterminer le typage de la fonction optimisée. Ainsi, si vous appelez la fonction f en écrivant une première fois f(4) et une seconde fois f(3.8), il y aura deux fonctions Numba créées, l'une acceptant un entier, l'autre acceptant un flottant.

L'efficacité est au rendez-vous. On peut facilement espérer un gain x20, voire plus si vous travaillez avec des int16 ou des int8.

Etrangement, un programme Python utilisant correctement Numba peut parfois devenir plus rapide que le même programme écrit en C ! En effet, Numba compile le code à la volée en utilisant les optimisations disponibles sur le processeur présent sur la machine alors qu'un code C est habituellement compilé en mode générique (x86) pour convenir à tout type de processeur.

Installation

Il vous reste à faire : pip install numba

Mise en place

```
from numba import jit
```

puis mettre le mot clef `@jit` la ligne précédent la fonction à optimiser

Contraintes

Le non-respect de ces contraintes met en danger le process d'optimisation :

1. Evitez les objets Python : liste / dictionnaire / class qui empêchent l'optimisation. Vous pouvez cependant utiliser les valeurs numériques x, y, score ou des tuples je crois. Cette contrainte concerne à la fois les paramètres en entrée, le code de la fonction et les valeurs retours.
2. Utilisez des tableaux Numpy, très proche du C, ils sont faciles à traduire pour Numba
3. Si vous appelez une fonction dans votre fonction @jit, elle doit être aussi @jit
4. Evitez d'utiliser des variables globales dans le code de la fonction car elles sont converties par Numba en valeur constantes correspondant à leurs valeurs lors du 1er appel, là où la compilation a eu lieu.

Portage

La plupart des fonctions de python/numpy ont été réécrites en Numba/C. Pour savoir si la fonction Numpy que vous utilisez a été portée sous Numba, vous pouvez vérifier dans la liste ici :

https://numba.pydata.org/numba-doc/dev/reference/numpysupported.html#other-functions

Il y a 95% de chance qu'elle le soit. A noter que les fonctions random ont été converties !

Exemples

Nous utilisons l'exemple suivant estimant la valeur de pi à partir de tirages au sort. Le temps consommé par le programme est proportionnel au nombre n de tirages. Nous mesurons le temps de calcul pour différentes valeurs de n correspondant à des puissances de 10.

```
from numba import jit
import random
import time

@jit
def monte_carlo_pi(nsamples):
   acc = 0
   for i in range(nsamples):
      x = random.random()
      y = random.random()
      if (x ** 2 + y ** 2) < 1.0: acc += 1  # test si le point est dans le cercle
   pi = 4.0 * acc / nsamples
   return pi

monte_carlo_pi(10) # le premier appel déclence la conversion Numba

for i in range(1,20):
   nb = 10**i
   print(nb)
   T0 = time.time_ns()      # prend le T0 au départ du calcul
   pi = monte_carlo_pi(nb)
   T1 = time.time_ns()      # prend le T à la fin du calcul
   print(pi)
   print( "durée en secondes : ",(T1-T0) / 10**9)   # T en nanosecondes
   time.sleep(1)            # force une pause sinon cela gèle les affichages
```

Voici les résultats obtenus :

n	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
Numba	~0	~0	~0	9 ms	93 ms	0.94 s	9,45 s	94,9 s
Python	~0	4 ms	38 ms	387 ms	3.87 s	38,3 s		
Gain				43	41	40.5		

D'autres exemples sur https://numba.pydata.org/

Evitez les rappels au noyau Python

Lorsque Numba voit une liste (objet python) dans le code, il peut passer en mode hybride. C'est à dire qu'il convertit le code en C et va conserver la liste comme un objet Python. Ainsi toute manipulation de la liste va déclencher un rappel au noyau Python, ce qui coûte très cher (c'est une sorte d'aller-retour Python/C). Cependant, cela marche et on peut quand même espérer un petit gain de rapidité de x2. Pour forcer Numba à ne pas rappeler le noyau Python en plein milieu du code C et ainsi convertir 100% du code de votre fonction, vous pouvez utiliser ce préfixe :

```
@jit(nopython=True)
```

Si Numba n'arrive pas à convertir 100% du code la fonction, alors il émettra une erreur.

Derniers conseils

Les messages d'erreur sont longs et peuvent paraître impressionnants, mais seules les lignes vers la fin sont réellement utiles.

Bon courage pour ceux qui veulent tenter cette aventure