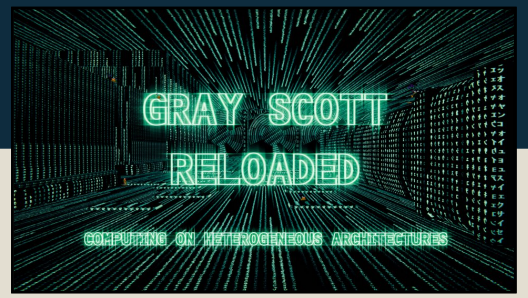
Python pour CPU

Jean-Marc Colley, Alice Faure

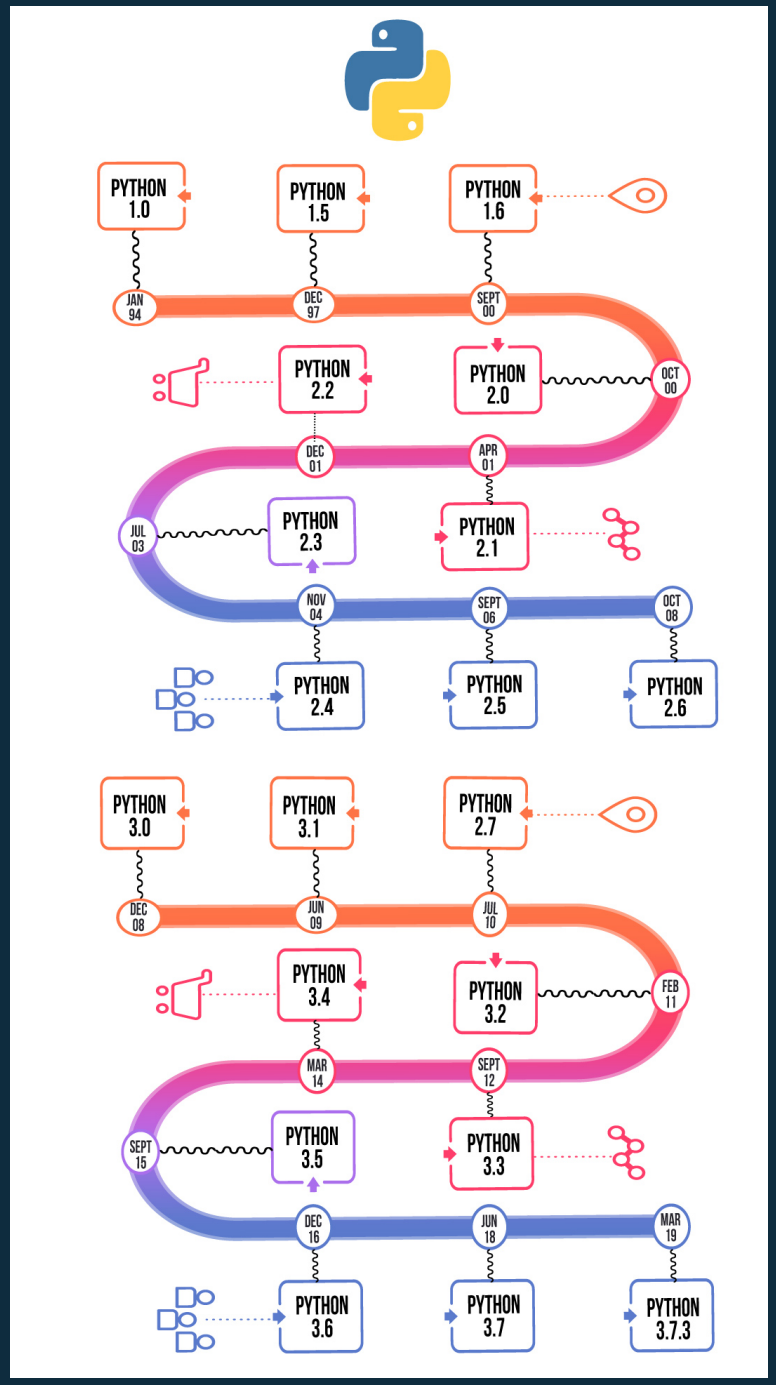


Présentation du langage Python

- Python un langage de programmation généraliste et open-source.
- Très flexible car dispose de nombreuses bibliothèques spécialisées.
- C'est l'un des langages les plus populaires, notamment dans le milieu de l'enseignement et la recherche.
- Très utilisé dans le domaine du machine learning, l'analyse de données, le développement web, de logiciels et d'applications, mais on peut également l'utiliser pour le calcul scientifique.

Principe

- Python est majoritairement utilisé comme un langage interprété.
- Si c'est le cas, il n'est pas compilé en code machine comme le C, le Fortran, etc. Au lieu de cela, le code source est compilé en byte code, un format intermédiaire qui est ensuite lu et exécuté ligne par ligne par l'interpréteur. Le code Python est donc plus lent qu'un code entièrement compilé, mais il est plus souple et plus facile à utiliser.
- Python est dynamiquement typé : l'utilisateur ne doit pas spécifier le type de la variable qu'il déclare.
- C'est aussi un langage multi-paradigme : il permet de faire de la programmation impérative, orientée objet, et fonctionnelle.



Historique

- 1989 : Guido van Rossum, un développeur néerlandais, invente le langage.
  But : obtenir un langage clair et facile à apprendre.
  La philosophie est résumée dans le Zen of Python.
  Il le nomme Python en référence aux Monty Python.

- 1991 : première publication du code

- 2000 : publication de la version 2

- 2008 : publication de la version 3, la dernière version majeure.

- La version 3.11 est 10 à 60% plus rapide que la version 3.10

Python pour le calcul scientifique

- La plupart des bibliothèques de calcul utilisées dans Python sont écrites en C et compilées (numpy, math, etc.), et le seul ralentissement dans l'exécution est lorsqu'elles sont appelées par le code Python.
- Un moyen simple d'avoir un code Python rapide est donc de passer la majeure partie du temps de calcul à utiliser ces bibliothèques optimisées. Il est également possible d'utiliser des compilateurs pour des sous-ensembles du langage Python pour accélérer le code Python.
- Dans le cours Python CPU de l'école Gray-Scott, nous apprendrons à écrire un code Python rapide et à l'optimiser pour le processeur.

Profilage – py-spy

py-spy est un profileur par échantillonnage pour les programmes Python. Il peut également profiler les codes multithreads et multiprocessus, et les extensions natives écrites en C/C++, sans avoir à modifier le code que l'on profile.

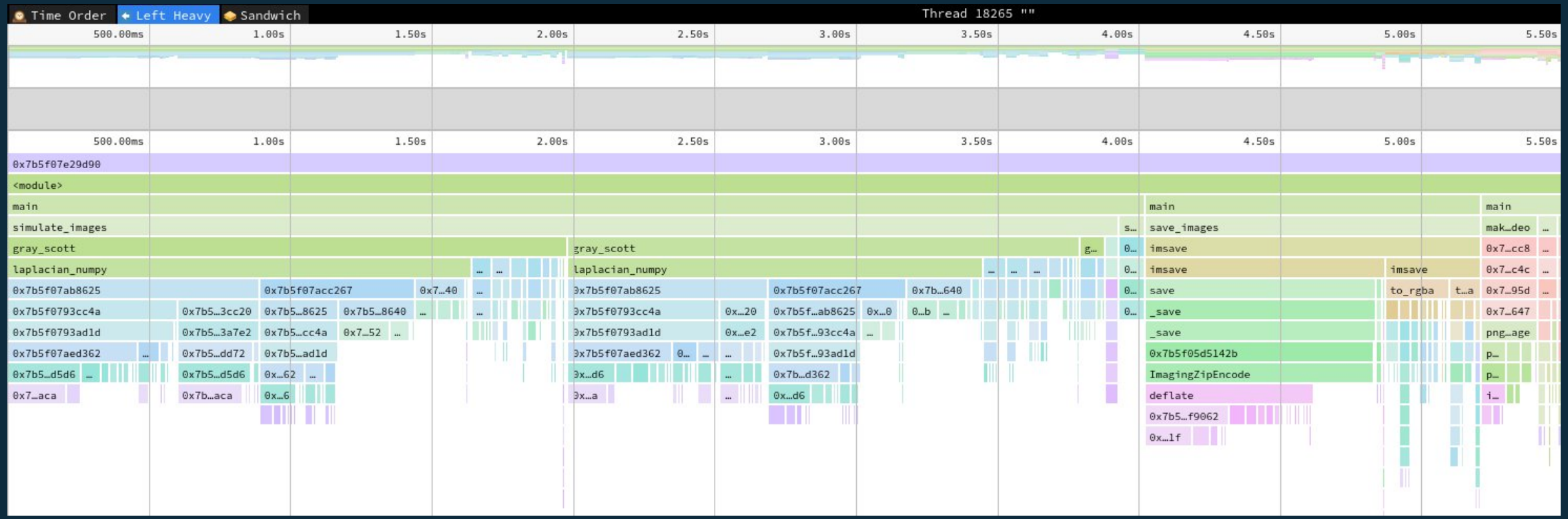
```
>>> pip install py-spy
>>> py-spy record -o profile.svg -n -- python gray_scott.py
```

Options pratiques :

- -o permet de spécifier le nom du fichier d'output
- -f permet de spécifier le format, par exemple speedscope
- -n permet de profiler les extensions natives
- -s active le support multiprocessus

Profilage – py-spy

```
>>> py-spy record -o profile.svg -f speedscope -n -- python gray_scott.py
```



Exemple de flamegraph généré par py-spy visualisé avec [speedscope]

Benchmarking – Module timeit

Ce module fournit une façon simple de mesurer le temps d'exécution de fragments de code Python. Il dispose d'une interface en ligne de commande ainsi qu'une interface Python.

Exemples tirés de la doc timeit

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
```

L'interface en ligne de commande détermine automatiquement le nombre de répétitions à effectuer.

Dans IPython (par exemple quand on utilise un notebook Jupyter), on peut utiliser la commande magique (le raccourci) intégrée %timeit.

Lien vers la doc d'IPython

Numpy présentation

- Manipulation de tableaux multidimensionnelles avec une bibliothèque de plus de 500 fonctions couvrant les domaines :
  - Algèbre linéaire et opérations terme à terme
  - Générateurs aléatoires
  - Fonctions mathématiques
  - Fast Fourier Transform
  - Recherche d'éléments, tri
- Les calculs utilisent les bibliothèques BLAS, LAPACK où la vectorisation SIMD est exploité, assurant des performances meilleures qu'en python natif
  - On peut voir sa config avec `np.show_config()`

Numpy présentation

```
In [3]: np.show_config()
Build Dependencies:
  blas:
    detection method: pkgconfig
    found: true
    include directory: /home/jcolley/miniforge3/envs/gs11_numpy/include
    lib directory: /home/jcolley/miniforge3/envs/gs11_numpy/lib
    name: blas
    openblas configuration: unknown
    pc file directory: /home/jcolley/miniforge3/envs/gs11_numpy/lib/pkgconfig
    version: 3.9.0
```

- Avec Scipy paru en 2006 et Matplotlib, ils ont progressivement remplacé Matlab et IDL dans nos labos
- C'est maintenant un standard

Numpy : classe ndarray

- L’objet centrale de la bibliothèque est ndarray (N-dimensional Array)
- Attributs importants :
  - .shape : « signature dimensionnelle », comme (10, 256,256)
  - .dtype : numpy défini ces propres types, comme : float32, int16
  - .nbytes : nombre d’octets occupés par le tableau
- Exemple de méthodes :
  - .reshape() : redéfinition de la signature dimensionnelle sans changement de taille
  - .astype() : changement de type
  - .copy() : retourne une copie

Numpy : broadcasting

- Notion de Numpy permettant de ne pas écrire une boucle python
  - => qui a priori est fatal pour les performances de calcul
  - **C'est donc une notion à connaître pour écrire une programme numpy performant**
- Le broadcasting le plus simple est un scalaire fois un tableau

```
In [1]: vec = np.array([1,2,3])

In [2]: 10 * vec
Out[2]: array([10, 20, 30])
```

Numpy sait qu'il doit multiplier par 10 tous les éléments du tableau

Plus difficile ... ajoutons des dimensions

Numpy : broadcasting

- Prenons maintenant un tableau d'images (10,256,256) que l'on veut « calibrer » individuellement avec un scalaire
- Chaque image a son coefficient de calibration que l'on stocke dans un tableau (10)

```
import numpy as np
ima  = np.random.uniform(0,50,10*256**2).reshape(10, 256, 256)
coef = np.random.uniform(0,50, 10)
coef*ima
```

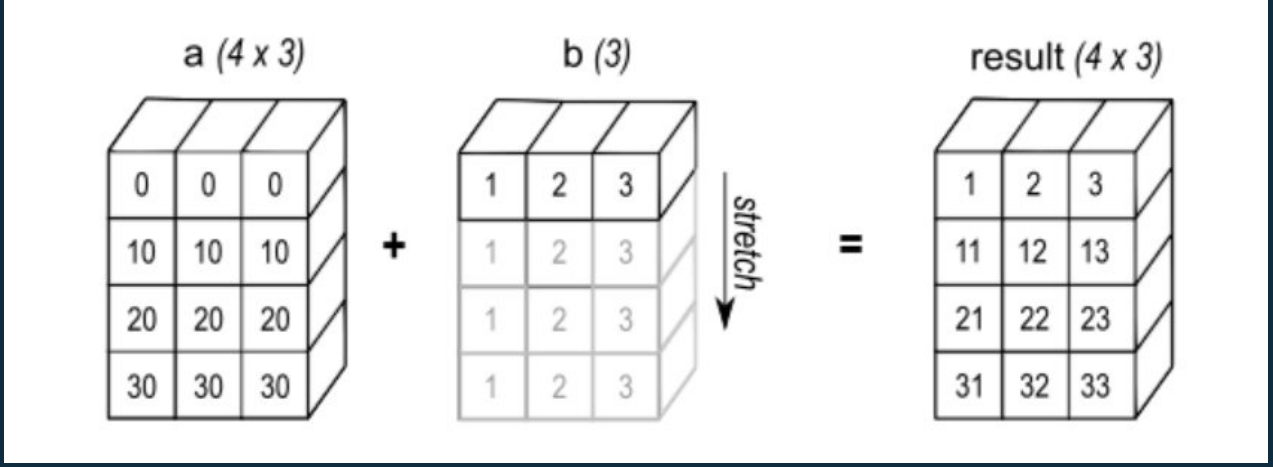
```
---> 10 a_coef*a_ima
ValueError: operands could not be
broadcast together with shapes (10,)
(10,256,256)
```

Aïe ça plante alors que c'est presque pareil que l'exemple précédent ...!

Numpy : broadcasting

- Les conditions du broadcast sont simples, il a lieu
  - Si les dimensions de « même rang » sont identiques : [C1]
  - Ou si l'une est égale à 1 : [C2]
- Par contre la notion de « même rang » est moins simple, car ce n'est pas la même donnée par l'attribut shape
- l'origine est à « droite », mais c'est logique avec cet exemple tiré de la doc numpy

Un tableau 1D est vu comme une ligne



Numpy : broadcasting

- Nous avons (10,256,256) * (10) qui plante, que faire ?
  - Demander de modifier la signature du tableau ...
  - Passer par une boucle, pour revenir au produit scalaire fois tableau qui marche
  - Modifier la signature dimensionnelle du tableau d'image
    - C'est à dire (256,256,10)*(10)
  - Modifier la signature dimensionnelle du tableau de coef.
    - C'est à dire (10,256,256) *(10,1,1))

Numpy : solution 1 « boucle naïve»

```
In []: def calib(ima, coef):
   ...:     for idx in range(coef.shape[0]):
   ...:         ima[idx] *= coef[idx]
```

```
In []: %timeit calib(ima, coef)
1.54 ms ± 410 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Numpy : solution 2 « moveaxis »

- On va modifier la signature du tableau d'image pour satisfaire la condition de broadcast, ie avoir (256,256,10)
- Fonctions modifiant l'ordre des dimensions
  - transpose(arr) ou arr.T : inverse l'ordre
  - moveaxis(arr, i_s, i_d) : insert i_s à la place i_d et décalage vers la gauche
  - swapaxis(arr, i1, i2) : échange les dimensions i1 et i2

```
aa.shape
(1,2,3)
```

```
np.moveaxis(aa, 0, 2).shape
(2,3,1)
```

Numpy : solution 2 « moveaxis »

- Mais Il est certainement nécessaire de rétablir le tableau d'image avec la signature d'origine :
  - => on faut ajouter un autre moveaxis après la multiplication

```
In []: %timeit np.moveaxis( np.moveaxis(ima, 0, 2) * coef , 2, 0)
```

```
274 µs ± 5.28 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

C'est beaucoup mieux que la version avec la boucle

Numpy : solution 3 « newaxis »

- On va modifier la signature du tableau de coefficient pour satisfaire la condition de broadcast sans toucher au tableau d'image, ie avoir (10,1,1)
- Il faut donc ajouter 2 dimensions supplémentaires, c'est le rôle de l'instruction np.newaxis, np.squeeze fait l'inverse

```
in []: a=np.array([1,2,3])
In []: a
Out[]: array([1, 2, 3])
```

Pas d'introduction de nouveau élément : une « ligne » est devenu une « colonne »

```
In []: a[:,np.newaxis]
Out[]: array([
       [1],
       [2],
       [3]])

In []: a[:,np.newaxis].shape
Out[]: (3, 1)
```

Numpy : solution 3 « newaxis »

- On augmente donc coef de 2 dimensions « à droite » avant de le multiplier
- Remarque : on peut écrire None à la place de np.newaxis

```
In [2]: %timeit   ima * coef[ :, None, None]

242 µs ± 7.79 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

C’est la meilleure performance des 3 méthodes !

Numpy : broadcasting validation

Vérifions que le résultat est le « même » avec les deux dernières méthodes, avec np.allclose() :

```
In []: r1 = ima * coef[:,None,None]

In []: r2 = np.moveaxis( np.moveaxis( ima, 0, 2) * coef, 2, 0)

In []: np.allclose (r1, r2)
Out[]: True
```

Numpy : broadcasting

- Sur cet exemple, la version avec newaxis sur le tableau de coef est la plus rapide
- Juste après, celle avec moveaxis sur la tableau d'image
- Et loin derrière, un facteur 6, la version avec la boucle

Utilisons des tableaux plus grand

Numpy : limitation du broadcasting

- La limitation du broadcasting se rencontre avec de gros tableaux, même exemple mais avec 10.000 images, soit un tableau de ~ 5GB

```
In []: ima.nbytes/1024**3
Out[]: 4.8828125

In []: %timeit ima*coef[:,None,None]
1.7 s ± 404 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In []: %timeit np.moveaxis(np.moveaxis(ima,0,2)*coef,2,0)
1.3 s ± 16.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In []: %timeit calib(ima, coef)
350 ms ± 2.93 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Numpy : limitation du broadcasting

- Les performances sont dans l'ordre inverse !
- La version avec la boucle est la meilleure car
  - Pas de consommation mémoire additionnelle en faisant des copies
  - elle ne déplace pas de données

Donc attention au passage à l'échelle, les tests de performances doivent être faits avec des données semblables à celles que vous devez traiter ou simuler.

Numpy : placement des données et performances

- Exemple avec le calcul de FFT sur un tableau de 3000 signaux 1D ~ 1GB

```
In [3]: tr3d_d= np.random.uniform(0,10,50*60*1024*50).reshape(50,60,1024*50)

In [4]: tr3d_g= np.moveaxis(tr3d_d, 2,0)

In [5]: tr3d_g.shape
Out[5]: (51200, 50, 60)

In [6]: res_d=np.fft.rfft(tr3d_d,axis=2)
In [7]: res_f=np.fft.rfft(tr3d_g, axis=0)

In [8]: np.allclose(np.moveaxis(res_g,0,2), res_d)
Out[8]: True
```

Numpy : placement des données et performances

- Exemple avec le calcul de FFT

```
In [5]: tr3d_g.shape
Out[5]: (51200, 50, 60)

In []: %timeit np.fft.rfft(tr3d_g, axis=0)
1.29 s ± 15.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In []: %timeit np.fft.rfft(tr3d_d, axis=2)
1.09 s ± 5.71 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- 20% de différence

Multithreading et multiprocessing

L'interpréteur Python de référence (CPython) fait appel à un GIL (Global Interpreter Lock) : c'est un verrou qui protège tous les objets qu'il manipule contre des accès concurrentiels.

Cela empêche le multithreading en Python (sauf pour les I/O, pour lesquelles le GIL est toujours relâché).

En général, pour faire du calcul parallèle en Python, il faut se rabattre sur le multiprocessing, par exemple avec le module dédié multiprocessing.

Le GIL est cependant relâché quand des librairies tierces sont appelées. Les backends de Numpy (BLAS, LAPACK, MKL...) font généralement usage des threads. Les calculs avec Numpy sont donc multithreadés par défaut.

Avec Numba, il est également possible de relâcher le GIL en passant l'option nogil=True (en mode nopython).

Numpy : multithreading

- Il faut considérer numpy en deux parties :
  - Le « frontend » API python
  - Le « backend » BLAS/LAPACK en c,c++/fortran
- Le multithreading repose uniquement sur le backend, en effet le GIL ne permet pas le multithreading
- Comme il y a plusieurs solutions de backends (mkl, openblas, netlib) et plusieurs versions, les fonctions multithreadés ne sont pas indiquées dans le doc numpy
- Contrôle du nombre de thread : par défaut tous les cœurs sont utilisés
  - On peut fixer la valeur avec la variable d'environnement associée à la bibliothèque multicore installée, comme :
    - **OMP_NUM_THREADS, MKL_NUM_THREADS**

Numpy : multithreading avec openblas

- C’est a prori le backend par défaut avec un simple :
  - `conda install numpy`
- Les opérations élément par élément se font sur 1 CPU
  - Numba permet de combler cette lacune
- Le produit matriciel np.dot() est lui multithreadé
- Liste indicative des fonctionnalités multithreadés avec un openblas récent, essentiellement ce qui se trouve dans linalg et qui peut s’écrire avec BLAS
  - https://superfastpython.com/multithreaded-numpy-functions/#List_of_Multithreaded_NumPy_Functions

Numpy : dépendance du backend

- Même version de numpy 1.26.4
- Mais avec un backend différent : OpenBLAS , mkl et 4 CPUs
  - FFT : x4 plus rapide pour mkl
  - np.dot(v,2*v) : x 1.5 plus rapide pour mkl
  - v*(2+v) : x 1.9 plus rapide pour mkl et multithreadé !
  - log(v*v) : x3.8 plus rapide pour mkl
  - max(v) : x1.4 plus rapide pour OpenBLAS
- Installation de numpy mkl, en python 3.10

```
conda install -c intel -c conda-forge intelpython3_full
```

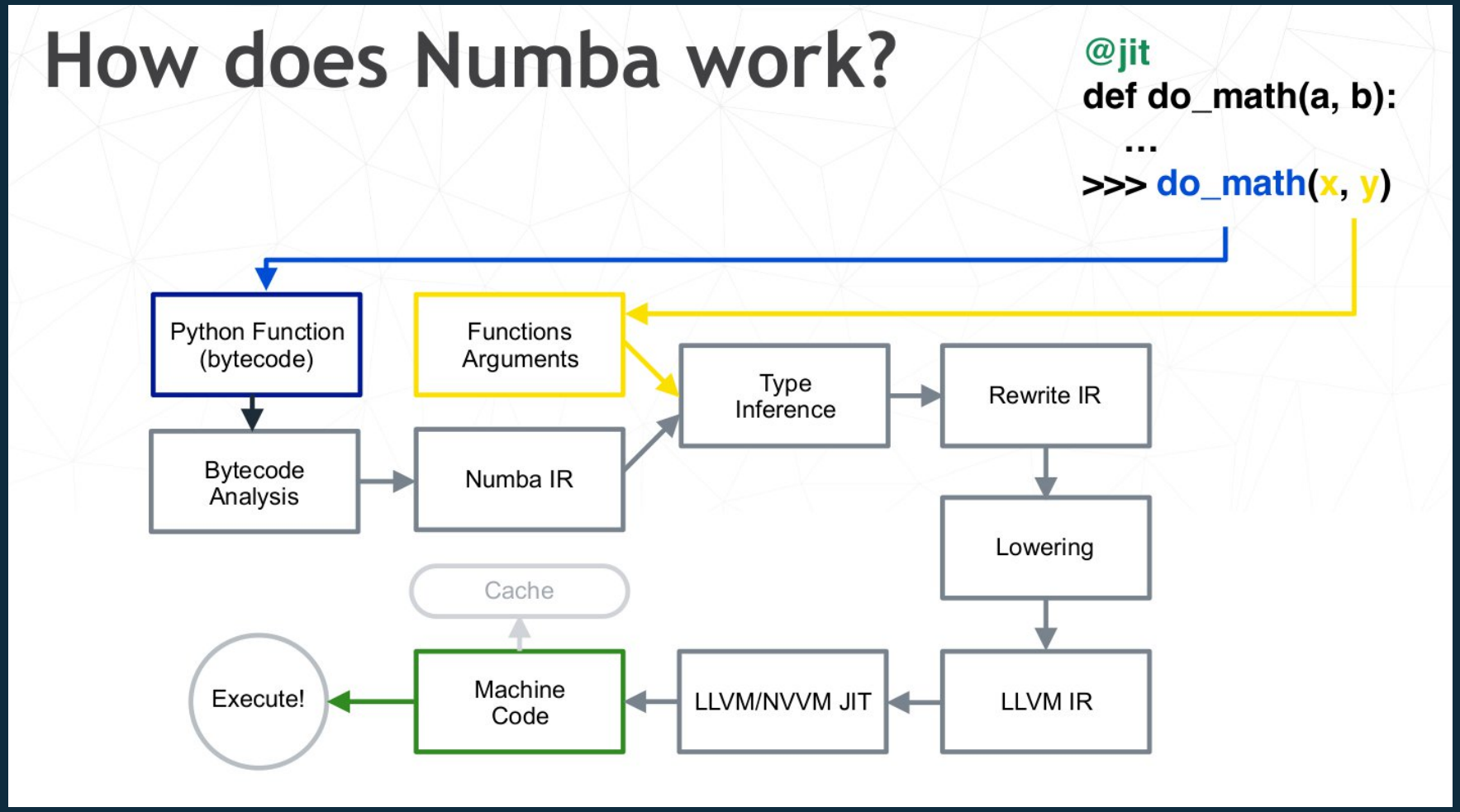
Intel DPNP : Data Parallel Numpy

- Clone de numpy pour les processeurs intel, utilisant comme backend
  - onemkl-sycl-blas, onemkl-sycl-lapack, onemkl-sycl-dft
  - 355 fonctions disponibles sur 542, liste dispo ici
- Comparaison pour quelques fonctions, c'est pas systématiquement mieux ! numpy avec backend MKL
  - Moins bien
    - FFT : 550MB
      - rfft : x 3.4
    - Fonction math
      - sin : x 1.4
      - exp :x 1.2
  - Mieux : DPNP plus rapide
    - générateur aléatoire : 500MB
      - uniform : x3 (mono coeur pour numpy :*)
      - normal : x 5   (*)
      - poisson : x 22 ! (*)
    - tri :
      - Max : x 1.8 (*)
      - Argsort : x 1.3 (*)

Numba - Présentation

- Numba est un compilateur à la volée (JIT, « just-in-time ») qui traduit les fonctions Python et Numpy en code machine optimisé.
- Il utilise la bibliothèque LLVM pour cela.
- Possibilité d'optimiser pour différentes architectures : CPU séquentiel et multi-threads, GPU.
- Les programmes Python compilés avec Numba peuvent approcher les performances des codes C et Fortran.
- Numba fonctionne surtout pour les codes qui utilisent des tableaux et fonctions Numpy, ainsi que des boucles. Le reste n'est pas supporté !

Numba - Compilation JIT



Tiré de cette [présentation de Numba]

Numba - Principe

Pour utiliser Numba il suffit de rajouter des décorateurs sur les fonctions ou les boucles que l'on veut optimiser.

Le décorateur @jit par exemple :

```python
from numba import jit

@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

Options fréquemment utilisées :

- nopython=True (le décorateur @njit est un alias pour @jit(nopython=True))
- parallel=True
- fastmath=True

```python
@jit(nopython=True)
def f(x, y):
    return x + y
```

Exemples tirés de la doc Numba

Numba – Multi-threading

Pour utiliser la parallélisation automatique avec Numba il suffit de :

- spécifier parallel = True
- utiliser prange au lieu de range dans les boucles for, pour indiquer qu'elles peuvent être parallélisées

```python
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the jit-decorator
    # the prange statement is equivalent to range
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

Exemples tirés de la doc Numba

Numba - Performances

- Numba est à utiliser quand on a un code fait de tableaux Numpy, avec des algorithmes numériques → Parfait pour le calcul scientifique !
- Il peut aussi servir à créer des fonctions Numpy « manquantes ».
- Pour mesurer les performances des fonctions optimisées avec Numba, mieux vaut utiliser le module timeit, qui mesure plusieurs itérations de l'exécution. Le temps passé à la compilation lors de la première itération est alors lissé.
- Le gain de performances peut aller de 2 (par rapport à Numpy) à 200 (par rapport au Python natif).

Numba - Performances

```
[10]: from numba import njit
      import numpy as np
```

```
[11]: def go_fast(a): # Function is compiled and runs in machine code
          trace = 0.0
          for i in range(a.shape[0]):
              trace += np.tanh(a[i, i])
          return a + trace
```

Exemples tirés de la doc Numba

```
[12]: x = np.arange(100).reshape(10, 10)
```

```
[13]: %timeit go_fast(x)
```

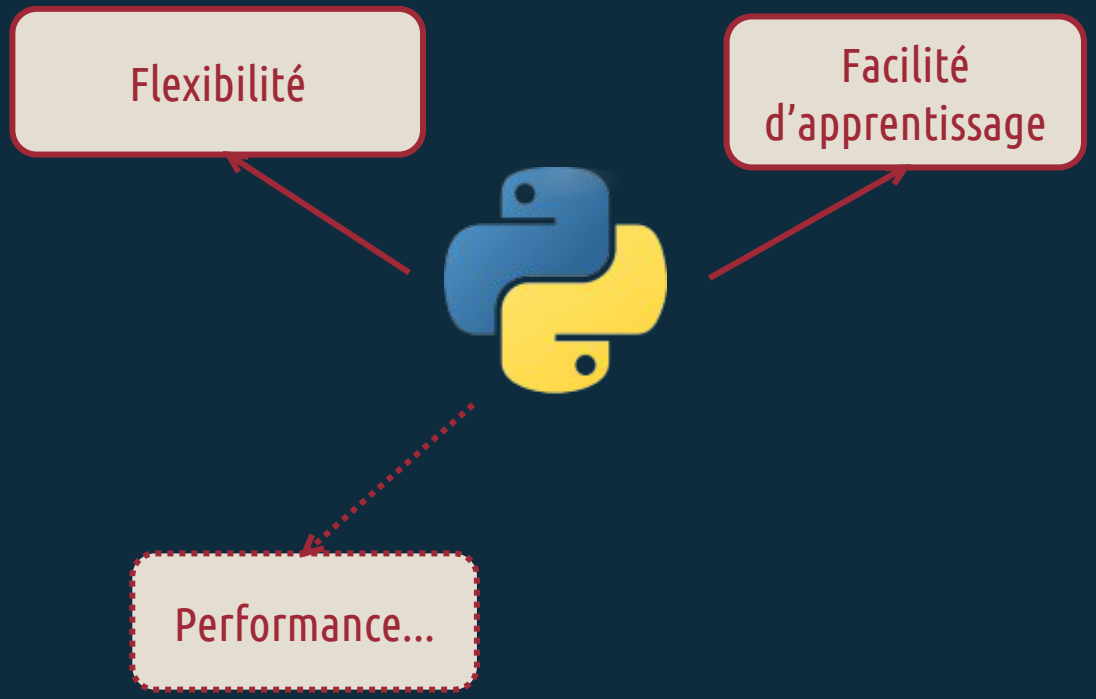
```
21.8 µs ± 1.15 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
[14]: @njit
      def go_fast(a): # Function is compiled and runs in machine code
          trace = 0.0
          for i in range(a.shape[0]):
              trace += np.tanh(a[i, i])
          return a + trace
```

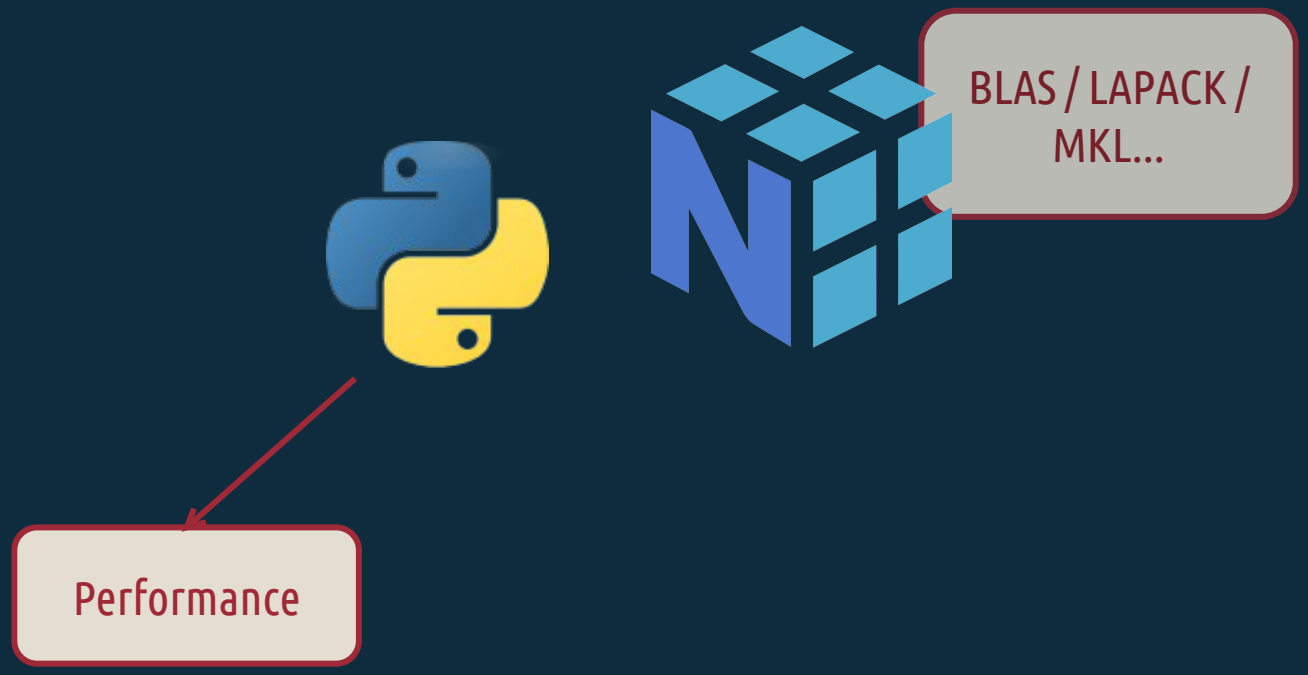
```
[15]: %timeit go_fast(x)
```

```
The slowest run took 6.66 times longer than the fastest. This could mean that an intermediate result is being cached.
3.36 µs ± 3.28 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

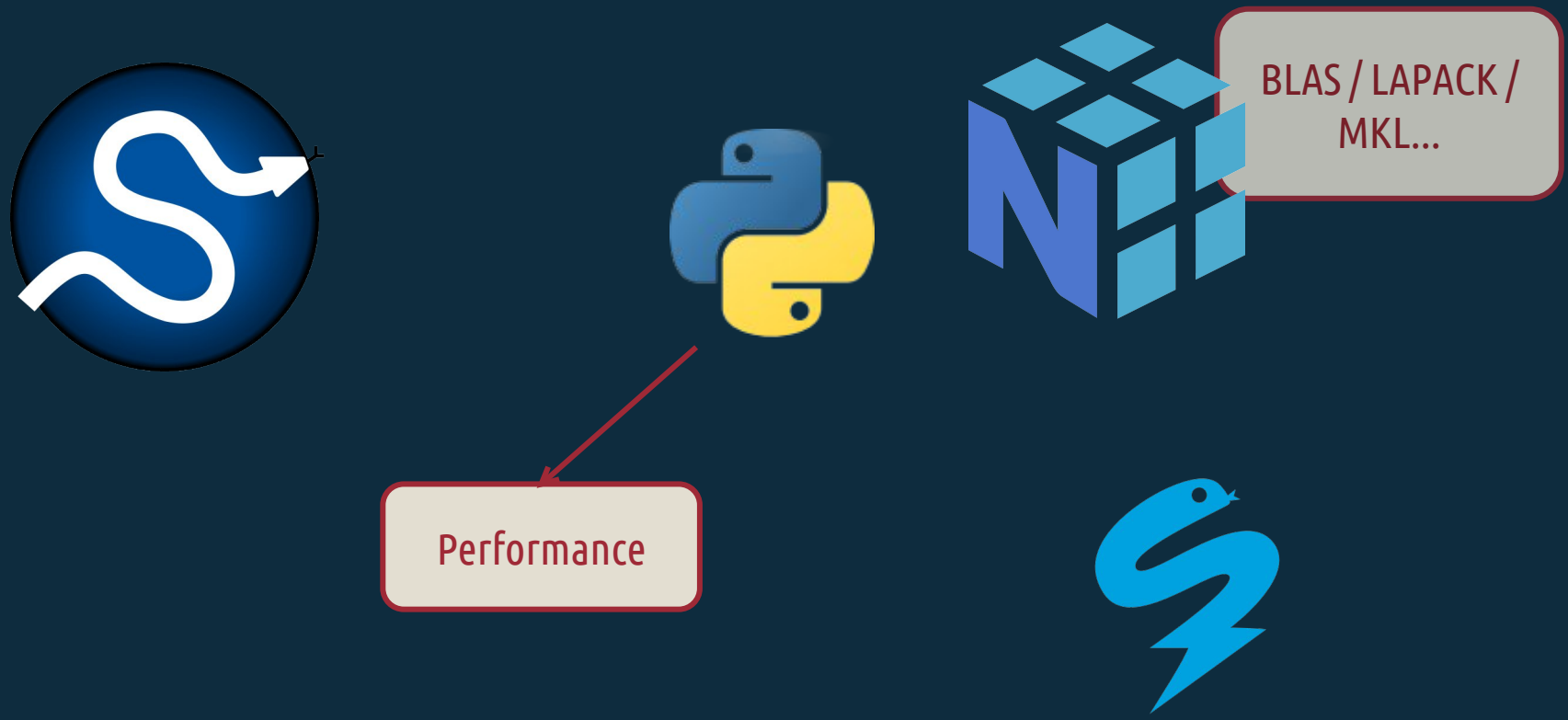
Conclusion



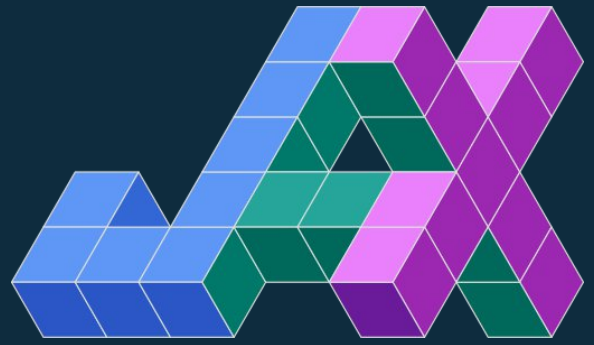
Conclusion



Conclusion



Conclusion



BLAS / LAPACK / MKL...

Performance

Conclusion