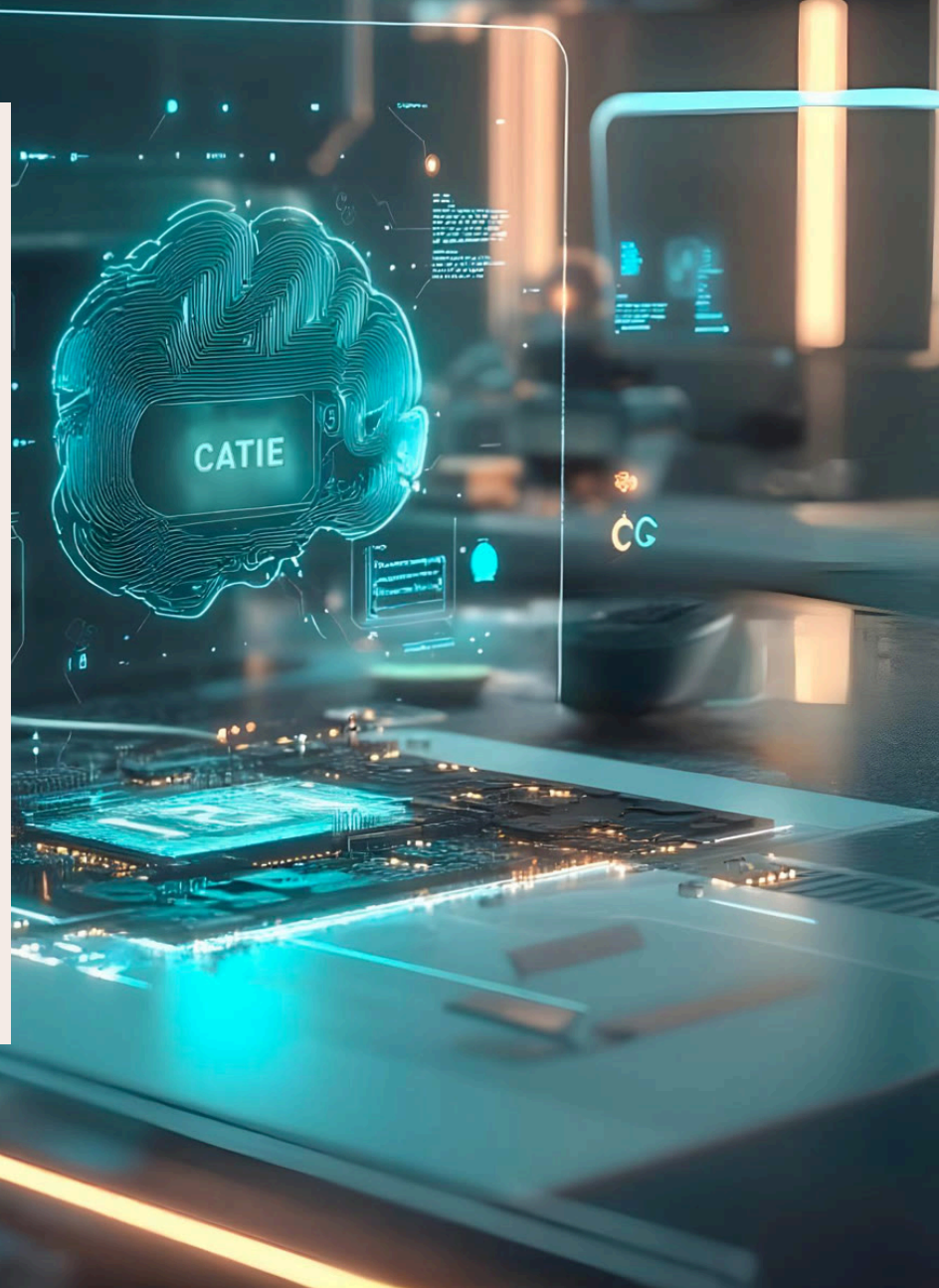




CATIE

Outils d'imagerie pour la
robotique
Python performant

13/10/2026



Objectif de la séance

Le fil rouge traite 200 images 320×240 (éclaircir, redimensionner, JPEG, statistique, archive) en **65 s**. Une caméra à 30 images/s laisse 33 ms par image, soit 6,7 s pour les 200 : il faut déjà aller 10 fois plus vite pour suivre le flux. On vise **moins d'une seconde**, pour garder de la marge (images plus grandes, machine embarquée plus lente), à résultat identique.

1. Mesurer

`cProfile` : où part le temps ?



2. Vectoriser

NumPy : plus de boucle
Python sur les pixels



3. Paralléliser

`concurrent.futures` :
tous les cœurs



4. TP

Fil rouge, étape 2

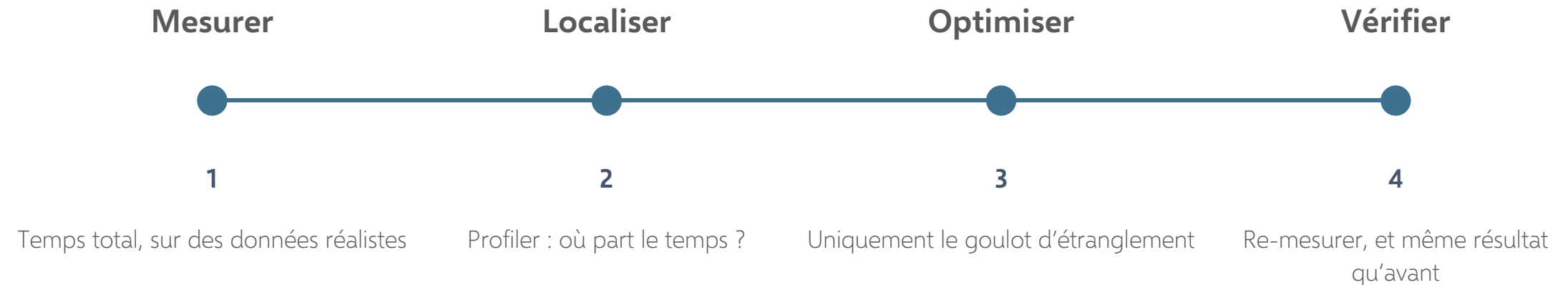
Les slides marquées Pour aller plus loin sont des compléments, non nécessaires au TP.



1. Mesurer avant d'optimiser

La démarche

I Premature optimization is the root of all evil. — Donald Knuth



■ Étape

Chronométrer : `time.perf_counter()` (monotone, haute résolution), pas `time.time()`. L'étape 1 le fait déjà : le décorateur `timed` affiche `run: 68.12 s`.

Optimiser une fonction qui prend 2 % du temps rapporte au mieux 2 %. L'intuition sur « ce qui est lent » est souvent fausse : on mesure.

Lire un profil cProfile

Le fil rouge (étape 1) sur 10 images :

```
$ uv run --with numpy --with pillow python -m cProfile -s cumtime pipeline.py data10 output
2456049 function calls in 4.406 seconds
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	4.131	4.131	pipeline.py:90(run)
10	0.000	0.000	4.128	0.413	pipeline.py:68(process)
10	3.632	0.363	3.942	0.394	operations.py:27(__call__)
2311187	0.314	0.000	0.314	0.000	{built-in method builtins.min}

- **tottime** : temps dans la fonction elle-même ; **cumtime** : fonctions appelées incluses
- 90 % du temps est dans **une seule** méthode, **Brighten.__call__**, appelée 10 fois, qui appelle **min** 2,3 millions de fois : c'est le goulot

Les outils de mesure

Pour aller plus loin

Outil	Pour quoi	Usage
<code>time.perf_counter()</code>	Chronométrer un bloc dans le code	différence de deux appels à <code>perf_counter()</code>
<code>timeit</code>	Micro-benchmark d'une expression, répété	<code>python -m timeit -s "import numpy as np" "np.zeros(10**6)"</code>
<code>cProfile</code>	Temps passé dans chaque fonction	<code>python -m cProfile -s cumtime pipeline.py</code>
<code>py-spy</code>	Profileur par échantillonnage, sans modifier le code, même sur un processus en cours	<code>uvx py-spy top -- python pipeline.py</code> <code>uvx py-spy record -o profile.svg --pid 1234</code>

- Mesurer plusieurs fois et garder le minimum : c'est le temps le moins perturbé par le reste de la machine
- Visualiser un profil : `uvx snakeviz profile.prof` (après `cProfile -o profile.prof`)



2. NumPy : calculer sur des tableaux

Le ndarray

```
import numpy as np
from PIL import Image

image = np.asarray(Image.open("photo.png"))
image.shape      # (240, 320, 3): h, w, channels
image.dtype      # dtype('uint8')
image.nbytes     # 230400: a single block
image[10, 20]    # array([182, 96, 41], dtype=uint8)
image[..., 0]    # red channel, shape (240, 320)
```

- Un bloc de mémoire **contigu**, un **type unique** pour tous les éléments : l'équivalent d'un `uint8_t[240][320][3]` en C
- Une liste Python, elle, est un tableau de pointeurs vers des objets indépendants
- Pillow lit et écrit les fichiers ; NumPy calcule. OpenCV, SciPy, scikit-image travaillent directement sur des **ndarray**

Vectoriser

```
def brighten(t, factor):  
    res = np.zeros(t.shape, dtype=np.uint8)  
    h, w, c = t.shape  
    for i in range(h):  
        for j in range(w):  
            for k in range(c):  
                v = int(t[i, j, k] * factor)  
                res[i, j, k] = min(v, 255)  
    return res
```

310 ms pour un tableau $240 \times 320 \times 3$

```
def brighten(t, factor):  
    return np.minimum(t * factor, 255) \  
        .astype(np.uint8)
```

0,9 ms : × 340, même résultat

Par élément, la boucle Python paie l'interprétation du bytecode, les vérifications de type, la création d'un objet `int`. NumPy exécute une boucle C compilée sur un type connu, souvent vectorisée par le processeur (SIMD).

Pas de boucle Python sur les éléments d'un tableau : on exprime le calcul sur le tableau entier, NumPy fait la boucle.

Les opérations courantes

Besoin	NumPy
Opération élément par élément	<code>t * 1.2, t + u, np.sqrt(t), np.minimum(t, 255)</code>
Réduction (somme, moyenne, max...)	<code>t.sum(), t.mean(axis=(0, 1))</code> (moyenne par canal)
Sélection par condition (masque)	<code>t[t > 200] = 200, np.where(t > 200, 255, t)</code>
Bornes, position d'un extremum	<code>np.clip(t, 0, 255), np.argmax(t)</code>
Changer de forme	<code>t.reshape(-1, 3), t.transpose(2, 0, 1), t[::-2, ::2]</code>

axis désigne la dimension réduite : sur une image (h, w, 3), `axis=(0, 1)` réduit hauteur et largeur et laisse une valeur par canal.

Les dtypes

```
a = np.array([200], dtype=np.uint8)
a + a                # [144]: overflow!
a.astype(np.int16) + a # [400]
a * 1.5              # [300.]: float64
```

```
shape = (1000, 1000, 3)
np.zeros(shape, np.uint8).nbytes # 3 MB
np.zeros(shape).nbytes           # 24 MB
```

- Les entiers de taille fixe **débordent sans prévenir** (modulo 256 pour `uint8`)
- Les opérations mélangeant types promeuvent vers le plus large (`uint8 * float → float64`)
- Choisir le type le plus petit suffisant : 8 fois moins de mémoire, donc plus de données dans le cache processeur

Calculer dans un type assez large, borner, puis revenir en `uint8` : c'est ce que fait **brighten**.

Broadcasting

Deux tableaux de formes différentes se combinent si, en comparant les dimensions **en partant de la fin**, chacune est égale ou vaut 1. La dimension 1 est « étirée » virtuellement, sans copie : comme appliquer le même tampon sur chaque ligne d'une feuille, sans photocopier le tampon.

```
m = rng.random((1000, 3)) # (1000, 3)
mu = m.mean(axis=0)      # (3,)
centered = m - mu        # (1000, 3)

weights = np.array([.2, .7, .1]) # (3,)
t = rng.random((240, 320, 3))
s = (t * weights).sum(axis=-1) # (240, 320)
```

```
a = np.arange(3).reshape(3, 1) # (3, 1)
b = np.arange(4)                # (4,)
a + b                           # (3, 4)
```

```
np.zeros((240, 320)) + np.ones(3)
# ValueError: operands could not
# be broadcast together with
# shapes (240,320) (3,)
```

Vues et copies

```
t = np.zeros((2000, 2000, 3), np.uint8) # 12 MB

region = t[100:1900, 100:1900] # view: 0.15 µs
region[:] = 255                 # also modifies t!
copied = region.copy()          # copy: 0.38 ms

evens = t[:, :2]                # view (regular step)
picked = t[[0, 5, 9]]           # copy (list)
masked = t[t > 128]             # copy (booleans)

np.shares_memory(t, region)     # True
```

- Une vue est une fenêtre sur les mêmes données, une copie est un duplicata
- Le **slicing** (`a:b:step`) crée une **vue** : aucune donnée copiée, coût quasi nul
- L'indexation par liste ou par masque crée une **copie**
- Modifier une vue modifie l'original : source de bugs, mais aussi d'économies
- `.copy()` explicite quand on veut un tableau indépendant
- `reshape`, `transpose`, `.T` renvoient des vues quand c'est possible



3. Parallélisme

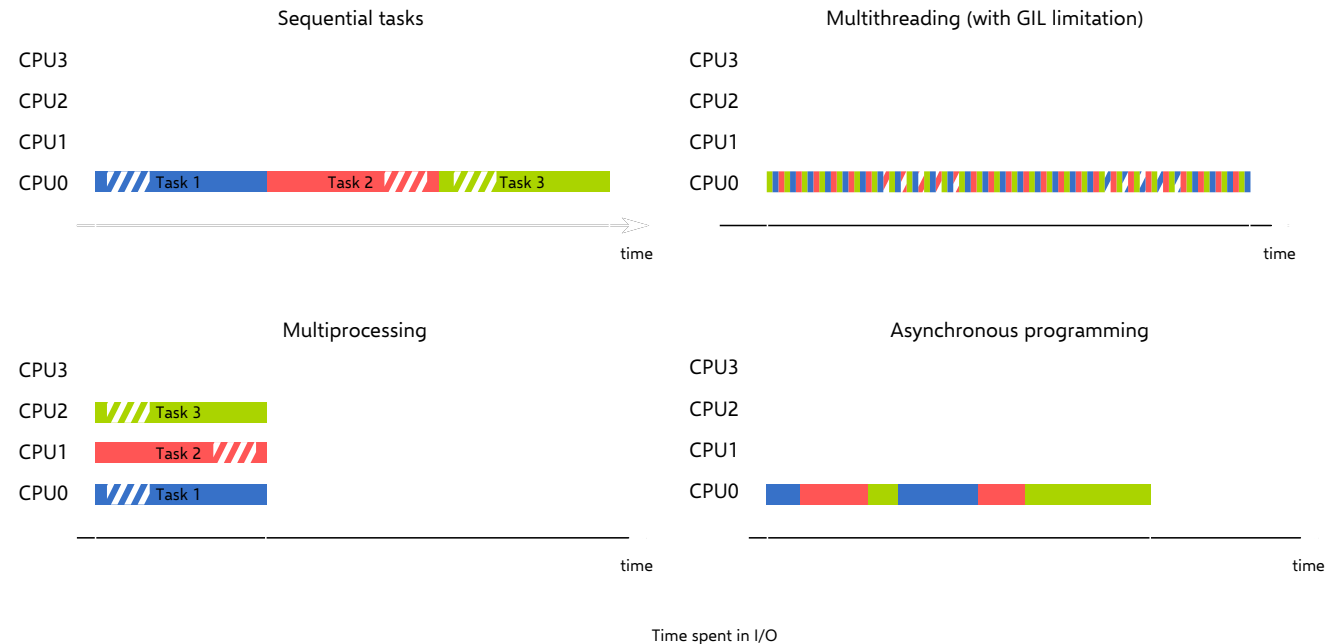
CPU-bound ou I/O-bound ?

	CPU-bound	I/O-bound
Le programme attend...	le processeur (calcul)	le disque, le réseau, un capteur
Exemples	calcul sur tableaux, compression, encodage	lecture de fichiers, requêtes HTTP, flux caméra
Symptôme	un cœur à 100 %	processeur peu utilisé, programme lent
Ce qui aide	vectoriser, puis plusieurs cœurs	faire autre chose pendant l'attente

Un vrai pipeline est souvent **les deux** : lire (I/O), calculer (CPU), écrire (I/O). Le profil dit quelle partie domine.

Le GIL

Le **Global Interpreter Lock** : dans CPython, un seul thread exécute du bytecode Python à la fois, même sur 16 cœurs. Il est **relâché** pendant les I/O et dans beaucoup de code C (NumPy, Pillow, OpenCV, zlib). Comme un bâton de parole : seul celui qui le tient parle, mais on le pose le temps d'attendre ou de calculer en C.



Threads ou processus ?

	Threads (ThreadPoolExecutor)	Processus (ProcessPoolExecutor)
Mémoire	partagée	séparée : un interpréteur et un GIL par processus
Parallélisme réel	seulement dans le code qui relâche le GIL	oui
Coût	faible	démarrage + sérialisation des données échangées
Idéal pour	I/O, calcul fait par NumPy, Pillow, OpenCV	calcul en Python pur sur plusieurs cœurs
Piège	accès concurrents aux données partagées	données volumineuses à transférer

Des threads, ce sont plusieurs personnes dans le même bureau ; des processus, plusieurs bureaux qui s'échangent des documents par courrier.

Un pipeline qui passe son temps dans NumPy et Pillow peut accélérer avec des threads malgré le GIL. Dans le doute, on mesure les deux.

concurrent.futures : une seule API pour les deux

```
from concurrent.futures import ProcessPoolExecutor, ThreadPoolExecutor, as_completed

def process(path: Path) -> Result: ...          # module-level function (picklable)

def main() -> None:
    paths = sorted(Path("data").glob("*.png"))
    with ProcessPoolExecutor(max_workers=8) as pool:      # or ThreadPoolExecutor
        results = list(pool.map(process, paths, chunksize=8))  # order preserved

    with ThreadPoolExecutor() as pool:
        futures = {pool.submit(process, p): p for p in paths}
        for future in as_completed(futures):             # as they complete
            result = future.result()                      # re-raises the worker's exception

if __name__ == "__main__":      # required with processes
    main()
```

Passer de threads à processus = changer une classe. Condition : des tâches indépendantes (ici, une image).

Le coût des processus

- Arguments et résultats traversent la frontière entre processus **sérialisés** (**pickle**) : aller-retour d'un tableau de 12 Mo $\approx$ **5 ms**
- Envoyer des **chemins** plutôt que des tableaux : chaque worker lit ses données lui-même
- **chunksize** : envoyer les tâches par paquets pour amortir la communication
- La fonction et ses arguments doivent être **picklables** : fonction de module, pas de **lambda**, pas de fichier ouvert
- Depuis Python 3.14, les processus sont lancés par **forkserver** sur Linux (**spawn** sous macOS et Windows) : le module principal est réimporté, d'où le `if __name__ == "__main__":`
- Démarrer des processus coûte cher : créer le pool **une fois**, pas par image
- Ce qui reste séquentiel (écriture de l'archive) borne le gain : 16 processus ne vont pas 16 fois plus vite (loi d'Amdahl)

Python sans GIL : le free-threading

Pour aller plus loin

- Python 3.13 : build expérimental sans GIL (PEP 703) ; Python 3.14 : build **officiellement supporté** (PEP 779) mais **optionnel** : `uv python install 3.14t`
- Code mono-thread un peu plus lent ; certaines extensions C réactivent le GIL au chargement

4 tâches de calcul en Python pur (boucle de 5 millions d'itérations), 4 workers :

Interpréteur	Séquentiel	4 threads	4 processus
Python 3.14 (avec GIL)	0,71 s	0,72 s	0,27 s
Python 3.14t (sans GIL)	0,80 s	0,22 s	0,30 s

Sans GIL, les threads valent les processus, sans le coût de communication. Pour le projet : build standard, sauf mesure qui prouve le contraire.

Flux continu : producteur / consommateur

Pour aller plus loin

```
import queue, threading

q = queue.Queue(maxsize=16)

def producer() -> None:
    for sample in sensor.read(): # I/O
        q.put(sample) # blocks if full
    q.put(None) # sentinel: end

threading.Thread(target=producer).start()
while (m := q.get()) is not None:
    process(m) # compute
```

- Pour un capteur, pas un lot de fichiers : l'acquisition continue pendant le calcul
- **maxsize** : si le calcul est trop lent, le producteur attend au lieu de remplir la mémoire (**backpressure**)
- Entre processus : `multiprocessing.Queue`
- Pour des milliers de connexions réseau : `asyncio`, un seul thread qui alterne entre les tâches en attente

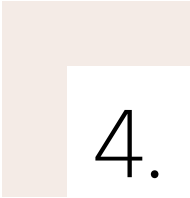
À retenir : mesurer et vectoriser

Besoin	Réflexe	En une ligne
Chronométrer	<code>perf_counter</code> , pas <code>time.time</code>	<code>start = time.perf_counter()</code>
Trouver le goulot	<code>cProfile</code> , trié par <code>cumtime</code>	<code>python -m cProfile -s cumtime ...</code>
Calculer sur une image	Opération NumPy, aucune boucle Python	<code>np.minimum(image * 1.2, 255)</code>
Éviter le débordement	Type large, borner, revenir en <code>uint8</code>	<code>np.clip(t, 0, 255).astype(np.uint8)</code>
Combiner des formes différentes	Broadcasting	<code>t * np.array([.2, .7, .1])</code>
Extraire une zone sans copier	Slicing (vue), <code>.copy()</code> si besoin	<code>region = t[10:-10, 10:-10]</code>

À retenir : paralléliser

Besoin	Réflexe	En une ligne
Calcul fait par NumPy, Pillow, ou I/O	Threads	<code>ThreadPoolExecutor()</code>
Calcul en Python pur	Processus	<code>ProcessPoolExecutor(max_workers=8)</code>
Garder l'ordre des résultats	<code>pool.map</code> , par paquets	<code>pool.map(process, paths, chunksize=8)</code>
Limiter les échanges entre processus	Envoyer des chemins, pas des tableaux	<code>def process(path: Path) -> Result:</code>
Lancer des processus sans surprise	Garde du module principal	<code>if __name__ == "__main__": main()</code>
S'assurer que rien n'a changé	Même sortie qu'avant	<code>cmp ref/stats.csv output/stats.csv</code>

Toujours dans cet ordre : mesurer, optimiser le goulot, re-mesurer, vérifier le résultat.



4. TP : fil rouge, étape 2

TP : consignes

Copiez votre `etape_1/` (ou le corrigé de la séance 1, sur le site) en `etape_2/` ; la sortie de l'étape 1 sert de référence.

1. **Mesurer** : temps total sur les 200 images, puis `cProfile` sur 10 images (`uv run ../depart/generate_images.py data10 10`). Notez les 3 fonctions les plus coûteuses.
2. **Vectoriser** ce que le profil désigne. Vérifiez que les sorties sont identiques : `cmp` sur `stats.csv`, `diff -r` sur `images/`.
3. **Paralléliser** avec `concurrent.futures` : options `--executor (sequential, threads, processes)` et `--workers`.
4. **Remplir le tableau** de la slide suivante sur votre machine.
5. **Expliquer** : pourquoi les threads accélèrent-ils ici malgré le GIL ? Pourquoi 16 processus ne vont-ils pas 16 fois plus vite ?

Bonus : `uvx py-spy top` pendant l'exécution ; essayer `uv run --python 3.14t`.

TP : tableau de mesures

Version	Référence (16 cœurs)	Votre machine
depart/	65 s	
etape_1/ (même algorithme, code propre)	68 s	
etape_2/ vectorisé, séquentiel	0,68 s	
etape_2/ + ThreadPoolExecutor	0,25 s	
etape_2/ + ProcessPoolExecutor	0,17 s	

200 images 320 × 240, Python 3.12. Le code propre n'est pas plus rapide : c'est la **vectorisation** qui fait × 100. Le parallélisme apporte ensuite un facteur 2 à 4, limité par ce qui reste séquentiel (écriture de l'archive, démarrage des workers).

À retenir

Mesurer

`perf_counter`, `cProfile`, `py-spy` : on optimise le goulot, rien d'autre

Paralléliser

`concurrent.futures`, `threads` ou `processus` selon la mesure

Vectoriser

Aucune boucle Python sur les éléments d'un tableau

Vérifier

Même résultat avant et après chaque optimisation

QUESTIONS ?