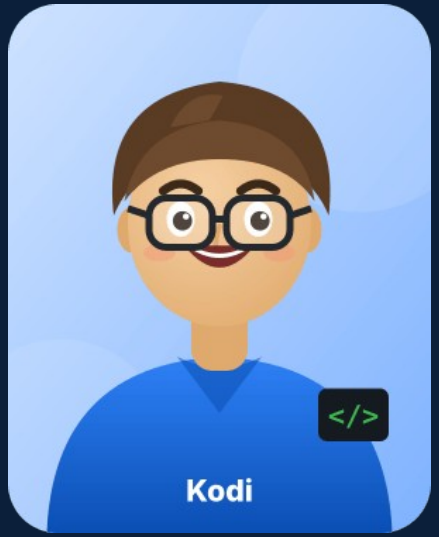
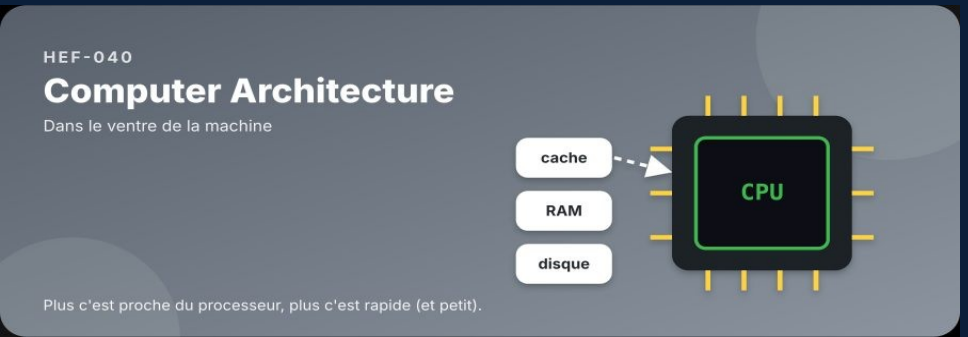
HEF-040 · MISSION 05 / 10

Cache et localité

Un programme rapide est un programme qui ré-utilise ce qu'il vient de toucher, et touche ce qui est voisin.



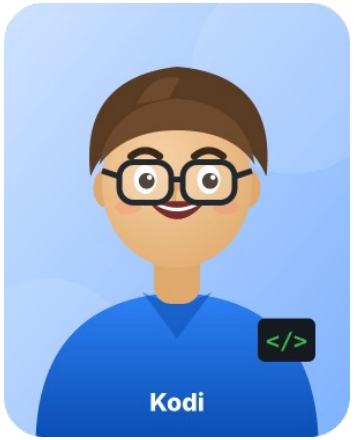


Badge : Maître du cache

75 min · débutant

L'histoire

Kodi parcourt une grande grille de données de deux façons : ligne par ligne, puis colonne par colonne. Le même calcul, les mêmes données. Mais le second parcours est beaucoup plus lent.



> *« Même nombre d'opérations, dit Sam. Mais le premier parcours profite du cache, le second le gaspille. Tout est une question de voisinage. »*

Tu comprends pourquoi l'ordre dans lequel on lit la mémoire change la vitesse d'un facteur 10 ou plus.

Ta mission

À la fin, je serai capable de…

1. expliquer le rôle du cache
2. définir ligne de cache, succès (hit) et échec (miss)
3. distinguer localité temporelle et localité spatiale
4. simuler un petit cache et compter les succès
5. écrire un parcours qui respecte la localité

Défi d'échauffement

Défi d'échauffement

Tu cherches des livres dans une grande bibliothèque. À chaque aller-retour au rayon (lointain), tu peux rapporter **8 livres voisins d'un coup** sur ton chariot. Tu dois lire les livres numéro 1, 2, 3, 4, 5, 6, 7, 8. Combien d'allers-retours ? Et si tu dois lire les numéros 1, 9, 17, 25, 33, 41, 49, 57 ? Compare.

Le concept

À quoi sert le cache ?

Le **cache** est une petite mémoire rapide placée entre le CPU et la RAM. Il garde des **copies** de ce qui a été lu récemment.

Quand le CPU veut une donnée :

- **succès (hit)** : elle est dans le cache → réponse en quelques cycles ;
- **échec (miss)** : elle n'y est pas → on va la chercher en RAM (~100 fois plus lent), et on la copie dans le cache.

La ligne de cache

Le cache ne copie pas un octet, mais un **bloc contigu**, la **ligne de cache** (typiquement **64 octets**). Lire une valeur ramène donc ses voisines.

Le concept

Deux types de localité

Localité	Idée	Exemple
Temporelle	ce qu'on vient d'utiliser sera probablement réutilisé bientôt	une variable de boucle
Spatiale	ce qui est proche en mémoire sera probablement utilisé bientôt	parcourir un tableau dans l'ordre

Les caches fonctionnent **parce que** les programmes ont de la localité.

Un tableau à deux dimensions

En mémoire, une matrice est rangée **ligne après ligne** (ordre dit *row-major*, comme en C ou NumPy).

Le concept

Un tableau à deux dimensions (suite)

```
matrice 3 x 4 en mémoire : [a00 a01 a02 a03 | a10 a11 a12 a13 | a20 a21 a22 a23]
```

- Parcourir **ligne par ligne** : accès voisins → peu de défauts de cache.
- Parcourir **colonne par colonne** : on saute d'une ligne entière à chaque pas → beaucoup de défauts de cache.

Simuler un cache

Un cache **à correspondance directe** : chaque ligne mémoire a une place fixe dans le cache (numéro de ligne modulo nombre de lignes du cache).

```
class Cache:
    def __init__(self, nb_lignes=8, taille_ligne=8):
        self.nb_lignes = nb_lignes
        self.taille_ligne = taille_ligne      # éléments par ligne de cache
```

Le concept

Simuler un cache (suite)

```
        self.contenu = [None] * nb_lignes     # numéro de ligne mémoire présente
        self.succes = 0
        self.echecs = 0

    def acces(self, adresse):
        ligne = adresse // self.taille_ligne
        place = ligne % self.nb_lignes
        if self.contenu[place] == ligne:
            self.succes += 1
        else:
            self.echecs += 1
            self.contenu[place] = ligne
```

🧠 ***Astuce*** *: pense « chariot de 8 livres ». Utilise tout ce que le chariot a rapporté avant d'aller chercher ailleurs.*

Exemple pas à pas

Problème : compter les succès et échecs pour un parcours ligne par ligne et colonne par colonne d'une matrice 32 × 32 (éléments rangés ligne après ligne).

```
class Cache:
    def __init__(self, nb_lignes=8, taille_ligne=8):
        self.nb_lignes, self.taille_ligne = nb_lignes, taille_ligne
        self.contenu = [None] * nb_lignes
        self.succes = self.echecs = 0

    def acces(self, adresse):
        ligne = adresse // self.taille_ligne
        place = ligne % self.nb_lignes
        if self.contenu[place] == ligne:
            self.succes += 1
        else:
            self.echecs += 1
            self.contenu[place] = ligne
```

Exemple pas à pas (suite)

```
N = 32

def parcours_lignes():
    c = Cache()
    for i in range(N):
        for j in range(N):
            c.acces(i * N + j)
    return c

def parcours_colonnes():
    c = Cache()
    for j in range(N):
        for i in range(N):
            c.acces(i * N + j)
    return c

for nom, f in (("lignes", parcours_lignes), ("colonnes", parcours_colonnes)):
    c = f()
    total = c.succes + c.echecs
```

Exemple pas à pas (suite)

```
    print(f"{nom:9s} échecs : {c.echecs:5d} / {total}  ({100 * c.echecs / total:.1
f} %)")
```

Résultat attendu : par lignes, un échec tous les 8 accès (12,5 %). Par colonnes, chaque accès tombe sur une ligne mémoire différente et le cache de 8 lignes est vite écrasé : presque 100 % d'échecs.
Conclusion : mêmes opérations, mais le parcours par lignes est bien plus rapide, car il exploite la localité spatiale.

À toi, avec aide

Avec la classe `Cache` de l'exemple, mesure le taux d'échecs d'un **parcours à pas fixe** d'un tableau de 4 096 éléments :

- pas 1 (tous les éléments) ;
- pas 2 ;
- pas 8 ;
- pas 16.

Que se passe-t-il à partir du pas 8 (taille de ligne = 8) ? Pourquoi ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pour un pas p : accéder aux adresses 0, p, 2p, 3p… jusqu'à 4 096.

Indice 2

Compte les échecs et divise par le nombre d'accès.

Indice 3

Quand le pas atteint la taille de ligne, chaque accès tombe dans une nouvelle ligne.

Réponse

```
class Cache:
    def __init__(self, nb_lignes=8, taille_ligne=8):
        self.nb_lignes, self.taille_ligne = nb_lignes, taille_ligne
        self.contenu = [None] * nb_lignes
        self.succes = self.echecs = 0
    def acces(self, adresse):
        ligne = adresse // self.taille_ligne
        place = ligne % self.nb_lignes
        if self.contenu[place] == ligne:
            self.succes += 1
        else:
            self.echecs += 1
            self.contenu[place] = ligne

for pas in (1, 2, 8, 16):
    c = Cache()
    for a in range(0, 4096, pas):
        c.acces(a)
    total = c.succes + c.echecs
```

À toi, avec aide · réponse

Réponse (suite)

```
    print(pas, f"{100 * c.echecs / total:.1f} %")
# 1 12.5 %, 2 25.0 %, 8 100.0 %, 16 100.0 %
```

À partir du pas 8, chaque accès tombe dans une ligne différente : plus aucun voisin n'est réutilisé, tous les accès sont des échecs.

À toi, en solo

Compare trois façons de calculer la **somme d'une matrice 64 × 64** en comptant les échecs de cache (avec un cache de 16 lignes de 8 éléments) :

1. parcours par lignes ;
2. parcours par colonnes ;
3. parcours **par blocs** de 8 × 8 (on traite un petit carré à la fois).

Laquelle est la meilleure ? Pourquoi le parcours par blocs aide-t-il même quand on doit accéder dans les deux directions (par exemple pour transposer une matrice) ?

À toi, en solo · réponse

Pistes de correction

```
class Cache:
    def __init__(self, nb_lignes=16, taille_ligne=8):
        self.nb_lignes, self.taille_ligne = nb_lignes, taille_ligne
        self.contenu = [None] * nb_lignes
        self.echecs = self.acces_total = 0
    def acces(self, adresse):
        self.acces_total += 1
        ligne = adresse // self.taille_ligne
        place = ligne % self.nb_lignes
        if self.contenu[place] != ligne:
            self.echecs += 1
            self.contenu[place] = ligne

N = 64

def lignes(c):
    for i in range(N):
        for j in range(N):
            c.acces(i * N + j)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def colonnes(c):
    for j in range(N):
        for i in range(N):
            c.acces(i * N + j)

def blocs(c, b=8):
    for bi in range(0, N, b):
        for bj in range(0, N, b):
            for i in range(bi, bi + b):
                for j in range(bj, bj + b):
                    c.acces(i * N + j)

for nom, f in (("lignes", lignes), ("colonnes", colonnes), ("blocs", blocs)):
    c = Cache()
    f(c)
    print(nom, c.echecs, f"{100 * c.echecs / c.acces_total:.1f} %")
```

Pistes de correction (suite)

Les lignes sont les meilleures pour une simple somme. Pour une transposition (lire par lignes, écrire par colonnes), le parcours par blocs garde un petit carré de données dans le cache et évite d'éjecter les lignes avant d'avoir fini de les utiliser.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Croire que tous les accès coûtent pareil	Un échec coûte ~100 fois un succès	Penser aux accès voisins
Parcourir une matrice dans le mauvais sens	Beaucoup d'échecs de cache	Parcourir dans l'ordre de stockage
Utiliser des structures dispersées	Listes chaînées : chaque maillon ailleurs en mémoire	Préférer les tableaux contigus pour de gros parcours
Optimiser sans mesurer	Le cache dépend de la machine	Mesurer avec des données réalistes

Mini-boss

Mini-boss

Laboratoire de localité. Construis un petit banc d'essai.

1. Étends `Cache` pour qu'il accepte une capacité (nombre de lignes) et une taille de ligne.
2. Simule trois programmes : (a) somme d'un tableau ; (b) somme d'une liste chaînée dont les maillons sont dispersés (adresses mélangées) ; (c) produit de matrices naïf (triple boucle) sur 16 × 16.
3. Pour chacun, trace le taux d'échecs selon la capacité du cache (4, 8, 16, 32 lignes).
4. Conclus : qui profite le plus d'un cache plus grand ? Qui ne profite pas de la taille de ligne ?

Mini-boss · réponse

Pistes de correction

```
import random

class Cache:
    def __init__(self, nb_lignes, taille_ligne=8):
        self.nb_lignes, self.taille_ligne = nb_lignes, taille_ligne
        self.contenu = [None] * nb_lignes
        self.echecs = self.total = 0
    def acces(self, a):
        self.total += 1
        ligne = a // self.taille_ligne
        place = ligne % self.nb_lignes
        if self.contenu[place] != ligne:
            self.echecs += 1
            self.contenu[place] = ligne

def somme_tableau(c, n=2048):
    for a in range(n):
        c.acces(a)
```

Mini-boss · réponse

Pistes de correction (suite)

```
def somme_chainee(c, n=2048):
    adresses = list(range(n)); random.Random(1).shuffle(adresses)
    for a in adresses:
        c.acces(a)

def produit_matrices(c, n=16):
    A, B, C = 0, n * n, 2 * n * n
    for i in range(n):
        for j in range(n):
            for k in range(n):
                c.acces(A + i * n + k); c.acces(B + k * n + j)
            c.acces(C + i * n + j)

for nom, f in (("tableau", somme_tableau), ("liste chaînée", somme_chainee), ("pro
   duit 16x16", produit_matrices)):
    print(nom, [f"{100 * (lambda c: (f(c), c.echecs / c.total)[1])(Cache(L)):.1f}%
   " for L in (4, 8, 16, 32)])
```

Mini-boss · réponse

Pistes de correction (suite)

Attendu : le tableau a ~12,5 % quel que soit le cache (il profite surtout de la ligne) ; la liste chaînée dispersée reste élevée tant que le cache ne contient pas tout ; le produit de matrices profite beaucoup d'un cache plus grand car il réutilise les mêmes lignes de B.

Quiz éclair

Quiz éclair

1. Un cache stocke :

A. les programmes installés
B. les mots de passe
C. des copies de données récemment utilisées

2. La localité spatiale signifie :

A. les données voisines sont souvent utilisées ensemble
B. une donnée récente sera réutilisée
C. la mémoire est proche du CPU

3. Parcourir une matrice rangée ligne par ligne est plus rapide :

A. colonne par colonne

Quiz éclair

Quiz éclair (suite)

B. ligne par ligne
C. au hasard

4. 🔁 *Révision mission 04* : Quel niveau de mémoire est volatile ?

A. le SSD
B. le disque dur
C. la RAM

5. 🔁 *Révision mission 03* : Le registre PC contient :

A. l'adresse de la prochaine instruction
B. le résultat du dernier calcul
C. la taille de la mémoire

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Dans ta vie, où profites-tu d'une « localité » (utiliser ce qui est à portée de main) ?
2. Pourquoi deux programmes qui font le même nombre d'opérations peuvent-ils avoir des vitesses très différentes ?

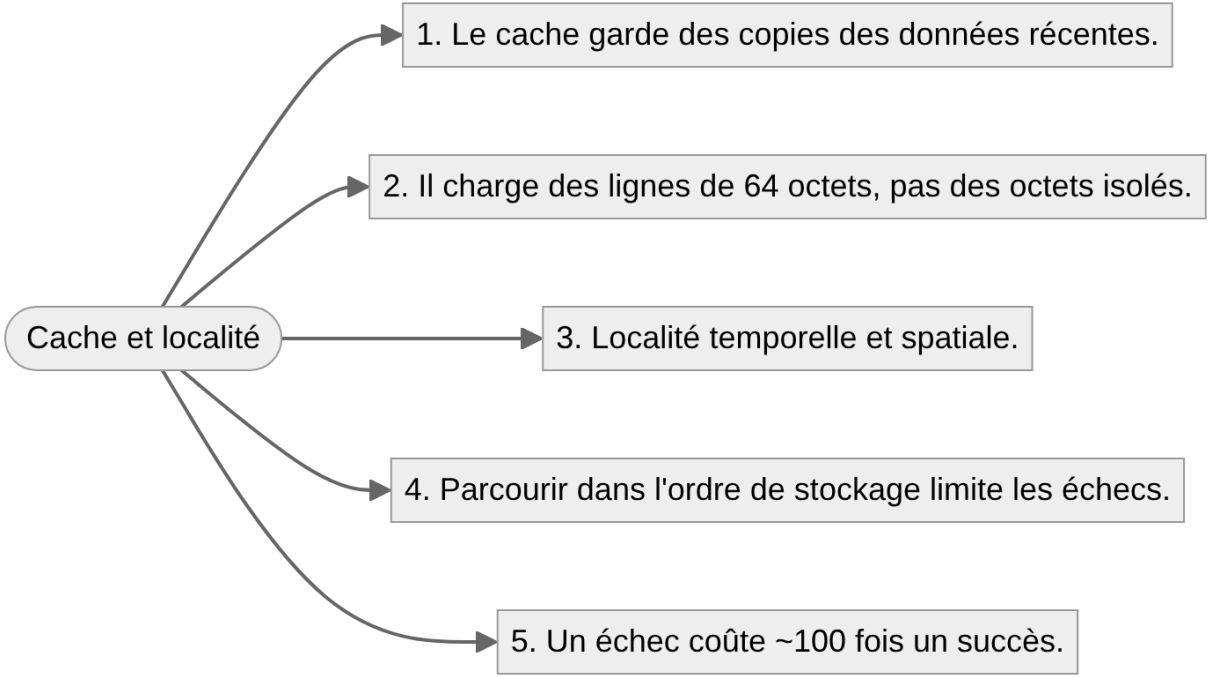
À retenir

À retenir

1. Le cache garde des copies des données récentes.
2. Il charge des lignes de 64 octets, pas des octets isolés.
3. Localité temporelle et spatiale.
4. Parcourir dans l'ordre de stockage limite les échecs.
5. Un échec coûte ~100 fois un succès.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître du cache

Prochaine mission : Stockage et entrées/sorties

