Lab 3
Mémoire cache

M. Briday

1 Objectif

L'objectif de ce TP est de se familiariser avec la hiérarchie mémoire (niveau de cache), et comprendre comment tirer profit du cache au niveau utilisateur (optimisation de code).

2 Simulateur de mémoire cache : `pycachesim`

Le simulateur `pycachesim` est en python 3 : `https://pypi.org/project/pycachesim/`. Son installation peut se faire directement avec le système de paquet [1] de python : `pip install pycachesim`

Il permet de simuler une hiérarchie complète avec plusieurs niveaux de cache. Dans un premier temps, nous définissons un seul niveau de cache :

```
from cachesim import CacheSimulator, Cache, MainMemory

mem = MainMemory()
# 8kB: 256 sets, 2-ways with cacheline size of 16 bytes
# with LRU replacement policy
cacheL1 = Cache("L1", 256, 2, 16, "LRU")
mem.load_to(cacheL1)
mem.store_from(cacheL1)
cs = CacheSimulator(cacheL1, mem)

#read one 32-bits value (4 bytes) at address 0x1234
cs.load(0x1234, length=4)

cs.print_stats()
```

1. Vous pouvez utiliser un environnement virtuel (`venv`) pour éviter d'installer des paquets de manière permanente. On crée l'environnement virtuel avec : `python3 -m venv venvPycacheSim` et on l'active avec `source venvPycacheSim/bin/activate`. la commande `deactivate` permet la désactivation.

Ce bout de programme minimal permet de définir le cache. Il y a un exemple sur la page du projet avec 3 niveaux de caches. On fait un accès en lecture et on affiche les statistiques :

```
CACHE *******HIT******** *******MISS******* *******LOAD******* ******STORE******* ******EVICT*******
 L1      0 (       0B)      1 (       4B)      1 (       4B)      0 (       0B)      0 (       0B)
MEM      1 (       4B)      0 (       0B)      1 (       4B)      0 (       0B)      0 (       0B)
```

Le nombre de hit/miss correspond au nombe d'appels à `cs.load(...)`. Comme c'est le premier accès, on a forcément un défaut de cache.

Attention : les accès à la mémoire pour charger la ligne de cache (les 16 octets) ne sont pas indiqués ! Les valeurs dans MEM correspondent aux accès requis pour lire une valeur (suite à un défaut de cache.)

3 1^{er} programme : lecture tableau

Configurer un cache de 2ko octets, à accès direct avec 8 octets/ligne. Simuler la lecture d'un tableau de 50 éléments (de 1 octet), à partir de l'adresse `0x10` :

```
for i in range(50):
  cs.load(0x10+i)        # Loads one byte from address
```

▷ quel est le taux de hit/miss. Était-ce prévisible ?

4 2^{e} programme : lecture matrice

En langage C, on déclare un tableau à 2 dimensions : `uint32_t tab[LINE_SIZE][COL_SIZE];`, où `LINE_SIZE` et `COL_SIZE` sont des constantes. L'accès mémoire à `tab[i][j]` est en fait un accès à l'élément `LINE_SIZE*i+j`. Dans ce cas précis, l'élément est un `uint32_t` de 4 octets, donc l'adresse mémoire correspondante est `tab+4*(LINE_SIZE*i+j)`.

Ainsi, la matrice `int mat[3][4]`, de 3 lignes et 4 colonnes a une représentation (index de chaque élément en mémoire) :

0	1	2	3
4	5	6	7
8	9	10	11

Le parcours d'un tableau de dimension 2 se fait à travers 2 boucles imbriquées :

— un parcours par colonne (boucle interne) puis par ligne (boucle externe) donne :
`0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11`.
— un parcours par ligne (boucle interne) puis par colonne (boucle externe) donne :
`0 - 4 - 8 - 1 - 5 - 9 - 2 - 6 - 10 - 3 - 7 - 11`.

▷ A priori, quel est le type de parcours qui sera le plus compatible avec l'utilisation d'une mémoire cache ?

Configurer un cache de 2ko octets, à accès direct avec 8 octets/ligne.

▷ simuler la lecture d'une matrice 128 lignes × 64 colonnes, le tableau étant situé à l'adresse 0x1234
— avec accès colonne puis par ligne
— avec accès ligne puis par colonne
En déduire les taux de *hit/misses* dans chaque cas. Expliquez la différence de comportement et justifiez le résultat particulier du 2e cas (schémas, ...)

Note : on peut faire plusieurs simulation dans le même script, à la suite. Dans ce cas, il suffit de réinitialiser les stats et invalider le cache :

```
cs.mark_all_invalid()
cs.reset_stats()
```

5 3^e programme : multiplication de matrice

Une multiplication de matrice se calcule sous la forme :

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{pmatrix}$$

avec :

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + a_{i2}b_{2j} = \sum_k a_{ik}b_{kj}$$

Pour chaque élément de la matrice (boucle sur i et sur j), il faut calculer une somme (3e boucle `for` imbriquée pour k).

On considère 3 matrices A, B et C dont l'adresse de base [2] est respectivement en 100, 200 et 300 (en décimal). On utilise un cache associatif à 2 voies, lignes de 16 octets, dont la taille totale est de 8ko. Chaque élément de matrice est un entier sur 8 bits.

De plus, pour simplifier, la somme partielle des c_{ij} et les compteurs de boucles sont optimisés (dans des registres) et il n'y a qu'un seul accès pour écrire la valeur finale pour chaque élément. Ainsi, pour le calcul de chaque élément, il faut n lectures dans la matrice A, n lectures dans la matrice B, et 1 écriture dans la matrice C.

Note : pour la simulation, il n'est nécessaire que de connaitre les emplacements mémoires et le type d'accès (`load` ou `store`), la valeur n'ayant aucune importance. Pour des raisons de mise au point, vous pouvez aussi travailler avec les tableaux en python pour vérifier que votre algorithme est fonctionnellement correct (ce sera certainement nécessaire pour la suite...).

▷ implementez la multiplication de matrice de taille 8x8.
▷ combien d'accès en lecture et en écritures sont nécessaires ?

2. soit l'adresse des éléments a_{00}, b_{00} et c_{00}

▷ quel est le taux de *hit*[3] ?

On augmente la taille des matrices pour que le cache soit un peu plus contraint :

— les données des matrices sont maintenant des entiers sur 32 bits ;
— les matrices ont 64x64 éléments, soit 16ko en mémoire
— les adresses de base des matrices A, B et C deviennent respectivement `0x100`, `0x4200` et `0x8300` ;

▷ modifiez votre simulateur pour prendre en compte ces nouvelles caractéristiques ;
▷ quel est le taux de hit ? expliquez (schéma) pourquoi il est si bas ?

6 4^e programme : multiplication de matrice par blocs

Une matrice peut être partitionnée en blocs. Par exemple :

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

où

$$A_{00} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, A_{11} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \ldots$$

Lors d'une multiplication, où A_{ij} et B_{ij} sont des blocs :

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}$$

où

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Le calcul revient à faire des multiplications de matrices de plus petites tailles (qui vont rentrer dans le cache). Ainsi, pour calculer C_{00} dans ce cas avec 2 blocs :

— on calcule d'abord $A_{00} \times B_{00}$, en mettant ce résultat intermédiaire dans C_{00}. Cette étape nécessite 3 boucles `for` imbriquées comme pour la multiplication de matrice standard.
— on calcule ensuite $A_{01} \times B_{10}$ et on ajoute le résultat au valeurs précédentes (une autre boucle `for` imbriquée ! !)
— et on reproduit ce schéma pour chaque bloc (en ligne et en colonne), soit encore 2 autres boucles `for` imbriquées ! !

Au final, l'algorithme est plus complexe (imbrication de 6 boucles `for`), mais on peut adapter les tailles des blocs pour profiter du cache.

il "suffit" de faire :

3. $\frac{nb_{hit}}{nb_{access}} = \frac{nb_{hit}}{nb_{hit}+nb_{miss}}$

```
pour chaque bloc (ligne bl, colonne bc)
  pour chaque blockId (bid)
    //on calcule le produit matriciel
    //du bloc de coordonnées bl.bc
    //il vaut:
```

$$C_{bl.bc} = \sum_{bid} A_{bl.bid} B_{bid.bc}$$

```
    pour chaque ligne l ,colonne c
      pour chaque k
        //on calcule le produit matriciel
        //à l'intérieur du bloc
        //ATTENTION aux indices pour le positionnement
        //matrice A (en nb de blocs) : ligne bl,  colonne bid
        //matrice B (en nb de blocs) : ligne bid, colonne bc
```

- ▷ implementer la simulation de la multiplication de matrice par bloc. Même configuration de cache que précédemment.
- ▷ déterminer le taux de hit pour les tailles de bloc en puissance de 2 (de 1 à 32). En déduire si l'opération semble rentable (vis à vis de l'utilisation de la cache) et indiquer la meilleures configuration de bloc pour le cache utilisé.