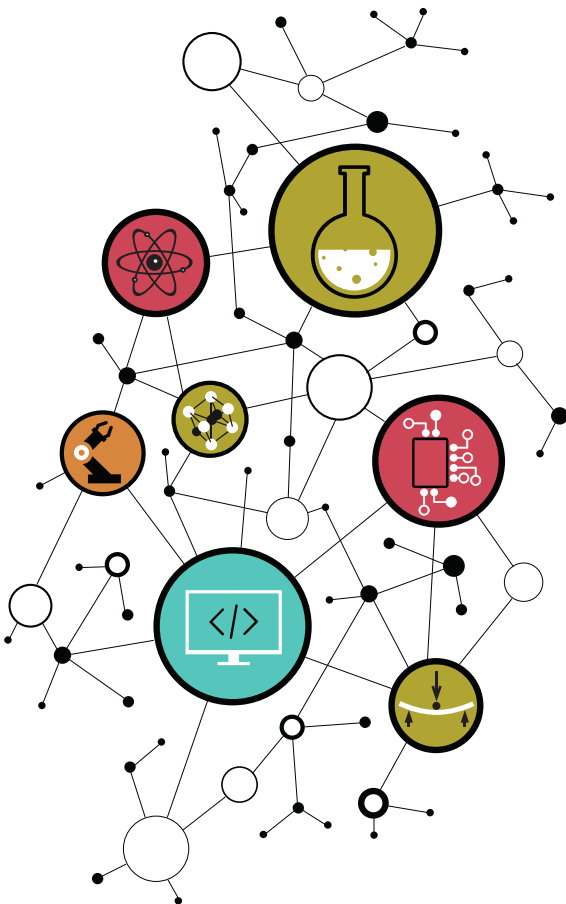


PARTIE 7

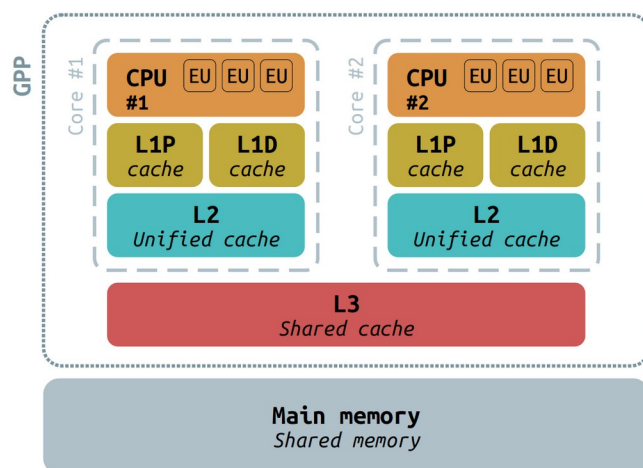
MÉMOIRES CACHE



I. Principes de localité

Les **mémoires cache** (ou parfois simplement **caches**) ne sont pas des zones mémoires accessibles par le système d'exploitation, car l'OS (ou le processus faisant une demande d'accès mémoire) n'a à sa disposition que l'espace de la mémoire principale constituée de la RAM et voire du swap.

Les caches se trouvent physiquement à l'intérieur du processeur et constituent une technique d'**accélération matérielle**, ou plus précisément d'accélération du temps d'accès à l'information en **dupliquant** ces informations au plus proche du CPU. Plus les informations sont spatialement proches des unités de traitement (EU, *Execution Unit*) du CPU, plus l'accès à ces informations est rapide. À ce titre on distingue plusieurs niveaux de cache, du plus lent (mais de plus grande capacité) au plus rapide (mais de capacité moindre) : la cache L3 propre au processeur, la cache L2 propre à chaque cœur (un cœur étant l'association d'un CPU et de ses caches attitrées) et les caches L1D et L1P (pour les *Data* et le *Program* respectivement).



Ces mémoires cache s'encapsulent : la L3 contenant une copie d'une portion de RAM, la L2 contenant une copie d'une portion de L3, la L1 contenant une copie d'une portion de L2. Si l'information demandée par le CPU est présente en cache L1, bingo : c'est un **cache hit** et le CPU dispose instantanément de l'information voulue. Si l'information demandée par le CPU n'est pas présente en cache L1, c'est un **cache miss** : on va alors la chercher en L2 pour la recopier en L1, voire la chercher en L3 si absente en L2, voire en mémoire principale en dernier recours.

Les avantages d'utiliser du silicium supplémentaire dans un processeur afin de gagner en temps d'accès à l'information s'explique par deux principes de localité :

- localité temporelle : une information juste manipulée va sûrement être réutilisée sous peu
 - d'où l'intérêt de garder l'information proche du CPU
- localité spatiale : les prochaines données à manipuler sont sûrement proches de la dernière
 - d'où l'intérêt de copier les informations par « bloc », qu'on appelle une ligne

Bien que les mémoires caches sont censées améliorer la vitesse d'exécution d'un programme, nous verrons qu'elles peuvent également être sources de contre-performances.

Complétez le schéma ci-dessus en y apposant les caractéristiques de votre processeur :

lscpu

II. Cache hit et cache miss

Placez-vous dans le répertoire `disco/cache/`.

Nous allons ici exécuter quatre exécutables très similaires : tous allouent (puis libèrent) un grand tableau de données, tous écrivent dans ce tableau grâce à une boucle `for`, et tous mesurent le temps d'exécution de cette boucle `for`. Ainsi le nombre d'instructions exécutées par ces quatre programmes est quasiment identique.

Les différences se font en deux points :

- les programmes `cache_v1` et `cache_v3` ne travaillent que sur les 10 premiers échantillons du tableau tandis que `cache_v2` et `cache_v4` travaillent sur l'intégralité du tableau ;
- les programmes `cache_v1` et `cache_v2` réalisent un grand nombre d'itération avec deux instructions C dans la boucle `for`, tandis que `cache_v3` et `cache_v4` réalisent 1000 fois moins d'itérations tout en faisant 1000 fois plus d'instructions dans la boucle `for`.

Pour tous ces programmes, vous pouvez jouer sur la macro constante `ARRAY_SIZE` pour « ajuster » le temps d'exécution de sorte à ce qu'il soit raisonnable. En revanche vous devez utiliser la même valeur pour les quatre fichiers : pensez donc à la mettre à jour partout si vous la modifiez !

Compilez chaque programme, puis exécutez-les avec l'outil d'analyse de cache de `valgrind`.

```
gcc cache_v1.c -o cache_v1
valgrind --tool=cachegrind --cachegrind-out-file=/dev/null ./cache_v1
```

Qu'observez-vous au niveau des temps d'exécution des programmes ?

Note : la différence entre `cache_v1` et `cache_v2` est plus probante avec une grande valeur pour `ARRAY_SIZE` mais votre machine va sacrément morfler avec le programme `cache_v4` ...

Qu'observez-vous au niveau accès aux mémoires cache (notamment les *cache misses*) ?

Liez les résultats de `valgrind` avec les quantités d'instructions et de données manipulées par chaque programme.

III. Cohérence de cache et false sharing

Le programme `threads.c` crée deux **threads**, qui accéderont en lecture et écriture à un tableau partagé. À notre niveau on considère que les *threads* sont deux sous-processus, les processus et *threads* seront étudiés en S8 en *Systèmes d'Exploitation et Réseaux*. La seule chose à savoir ici est que les *threads* vont s'exécuter simultanément, chacun sur un CPU différent.

Les *threads* `thread_write_first_half` et `thread_write_second_half` se séparent le tableau en deux moitiés égales, tandis que les *threads* `thread_write_even` et `thread_write_odd` gèrent chacun un échantillon sur deux, respectivement les échantillons pairs et impairs.

Compilez et exécutez le programme `threads.c`.

```
gcc threads.c -lpthread -o threads
./threads
```

Comment expliquer l'écart de performances entre les deux versions, alors que la quantité de données traitées et le nombre d'instructions sont très similaires ?

Sur les processeurs d'architecture Intel x86 ou x64, la taille d'une ligne de cache est généralement de 64 octets. Dans notre exemple, cela correspond à 16 cases du tableau ($16 * \text{sizeof(int)} = 64$ octets).

Si le CPU0 copie 16 cases du tableau dans sa cache L1D et que le CPU1 copie 16 autres cases dans sa cache L1D, chaque CPU peut travailler indépendamment sur son lot de données. C'est grosso modo ce qui se passe avec les versions `first_half` et `second_half`.

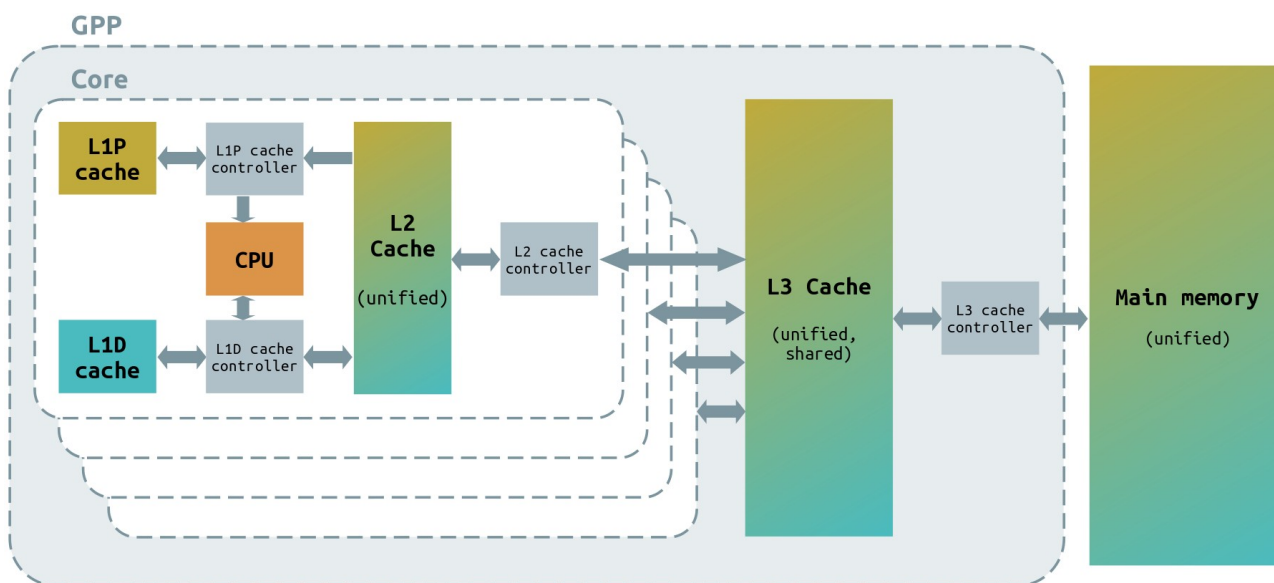
En revanche si le CPU0 copie 16 cases du tableau dans sa cache L1D et que le CPU1 copie les 16 mêmes cases du tableau dans sa cache L1D, alors chaque CPU doit vérifier si les copies présentes dans sa cache L1D sont à jour, puisqu'elles pourraient avoir été modifiées par l'autre CPU. C'est ce qui se produit avec les versions `even` et `odd`, où les données manipulées par les deux CPU sont **adjacentes** en mémoire.

Cet exercice met donc en avant un problème de **cohérence de cache**, ce qui signifie que les données contenues dans une cache L1 ne sont pas forcément les données à jour. Ce cas précis est même un cas particulier de cohérence appelé le **false sharing** (faux partage), où deux CPU accèdent en même temps à des données situées sur la même ligne de cache (puisque les informations en mémoire principales sont adjacentes).

IV. Synthèse

Bien que l'objectif des exercices précédents est de montrer les problèmes de performances liés aux mémoires cache (et donc ce qu'on observe sont les dégradations de performances des programmes), il faut quand même mettre ces résultats en perspective avec le taux de réussite d'accès à l'information.

En effet dans le premier exercice, même si le nombre de *cache misses* peut augmenter d'un facteur 1000, le taux de *cache misses* reste quant à lui très faible. Ces performances sont notamment dues à la politique de gestion des mémoires caches, assurant le chargement des données depuis la mémoire principale vers une cache et la restitution en mémoire principale par la suite. Ces politiques sont gérées par du matériel appelé contrôleur de cache et présent à chaque niveau de mémoire cache à l'intérieur du processeur.



Le deuxième exercice montre lui que le choix de l'algorithme impactera sensiblement les performances du programme. Dans les faits les développeurs haut-niveau ou orientés application n'auront pas à se soucier de la cache en particulier, ni même du matériel en général. En revanche les développeurs d'algorithme (au sens traitement mathématiques appliqués sur une grande quantité de données) devront quand à eux s'intéresser aux couches matérielles du système (les mémoires cache, mais aussi les différentes unités d'exécution du CPU, les étages du pipeline, ...) puisque ces éléments seront fortement sollicités pendant l'exécution de l'algorithme. Un léger gain (ou au contraire un léger délai) s'appliquant à une donnée, se répercutera sur la totalité des données et donc générera un fort gain (ou forte dégradation) à l'ensemble du programme.

En bref, les caractéristiques des mémoires caches (temps d'accès, capacité, taille d'une ligne, politique de remplacements des lignes ...) sont autant de paramètres permettant d'améliorer les performances d'un algorithme en terme de temps d'exécution. Même si ce mécanisme matériel est transparent pour le développeur (au même titre que le *pipeline* par exemple), il faut dans certains cas être capable de descendre au niveau *hardware* pour comprendre ses forces et (parfois) ses faiblesses.

