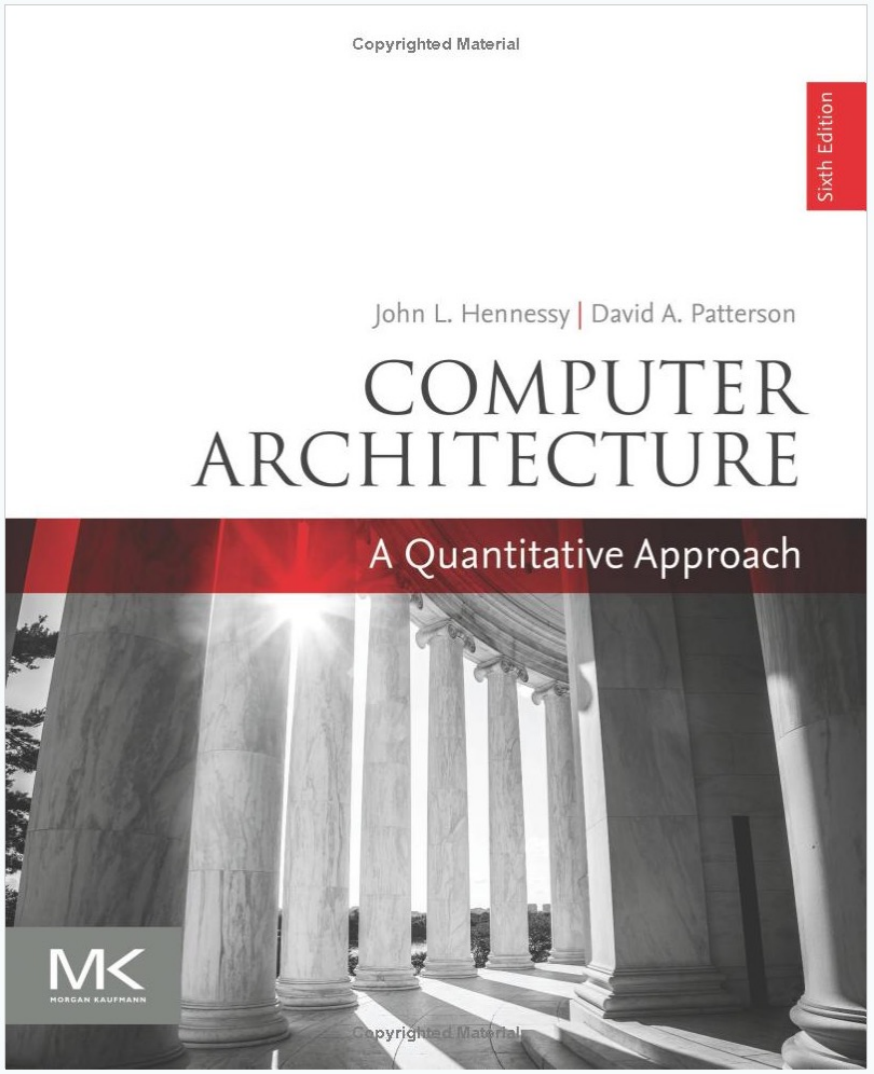
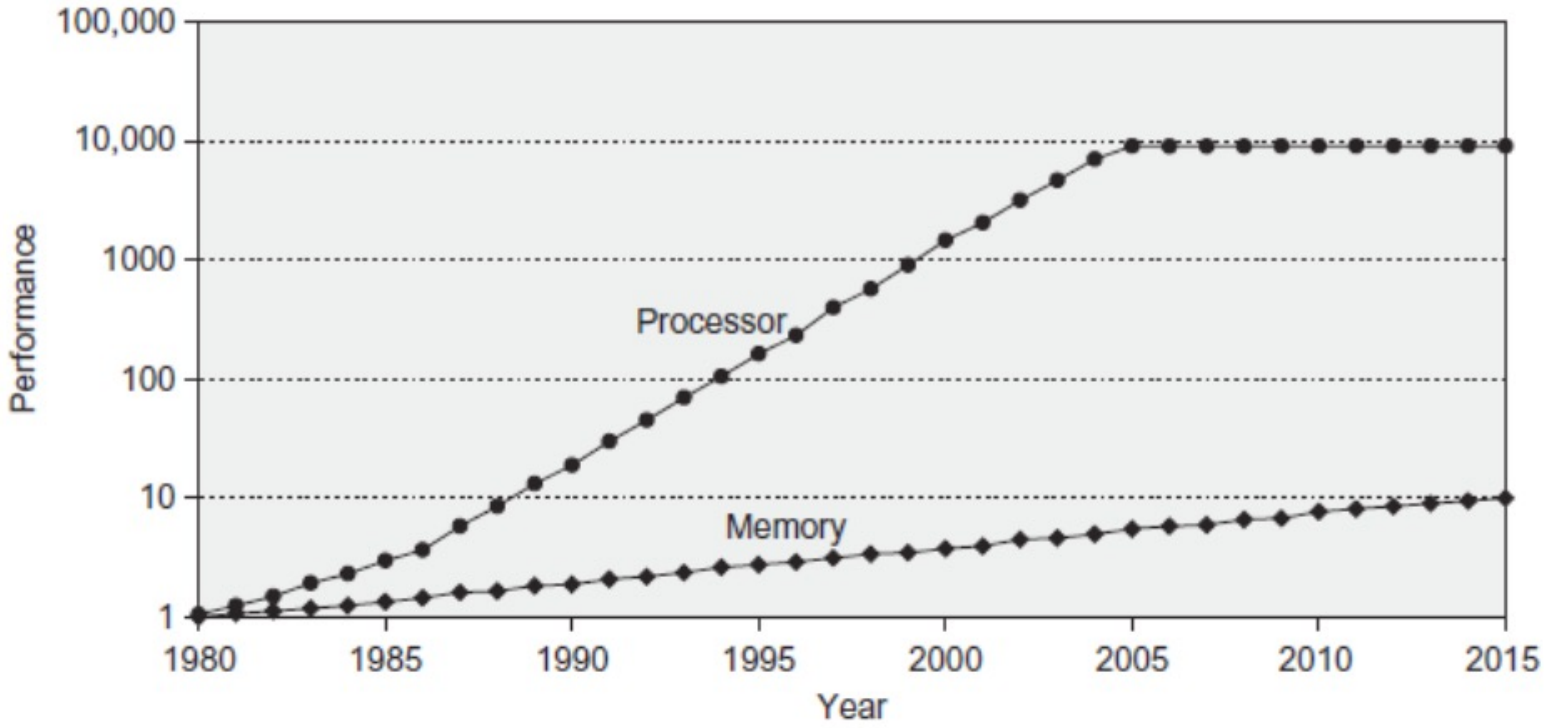
La mémoire cache

Raymond Namyst
Pierre-André Wacrenier



Pourquoi a-t-on besoin des caches ?

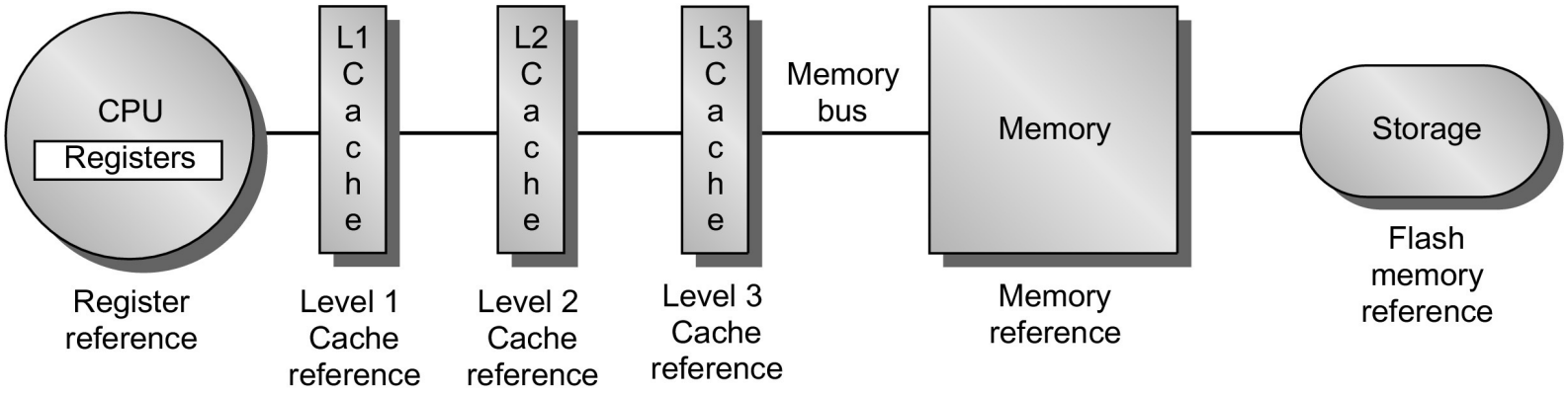


De l’utilité d’une mémoire de proximité

Comment obtenir les performances promises par les processeurs ?

- Exploiter une hiérarchie mémoire
  - Plus une mémoire est petite plus son temps de réponse peut être court

Hiérarchie mémoire d'un ordinateur



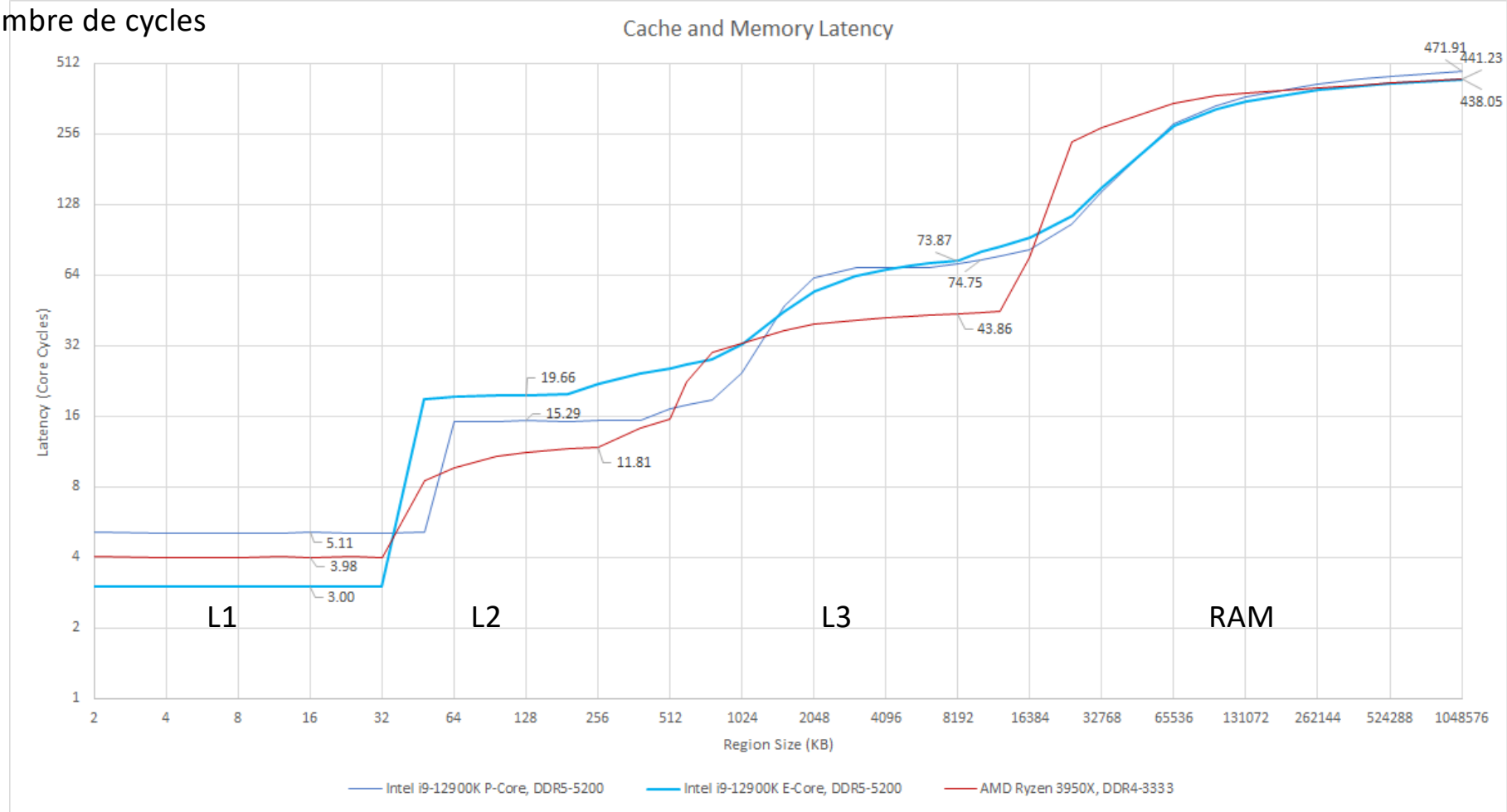
Laptop	Size:	1000 bytes	64 KB	256 KB	4-8 MB	4–16 GB	256 GB-1 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS
Desktop	Size:	2000 bytes	64 KB	256 KB	8-32 MB	8–64 GB	256 GB-2 TB
	Speed:	300 ps	1 ns	3–10 ns	10–20 ns	50–100 ns	50-100 uS

(B) Memory hierarchy for a laptop or a desktop

Les caches sont souvent inclusifs (i.e. le cache L2 inclut le contenu du L1)

Latence intel alderlake & amd ryzen 3950x

Nombre de cycles



Comment Placer les données les plus *importantes* dans les mémoires les plus rapides

- Donner le contrôle au programmeur ?
  - Variables *register* du langage C
  - Allocateurs spécifiques au matériel (GPU, Accélérateurs)

- Assister matériellement le programmeur (de façon transparente)
  - Comportement régulier des programmes (boucles)
  - Localité temporelle
    - Règle 90/10 « 90% de l'exécution se déroule dans 10% du code »
    - Accès répétés à des mêmes variables, de façon proche dans le temps
  - Localité spatiale
    - Concentration des accès sur des données proches
      - Structures, tableaux, variables locales

- **Favoriser l'accès séquentiel à la mémoire**

Exemple

```
// Tile computation
int invert_do_tile_default (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);

  return 0;
}

int invert_do_tile_bad (int x, int y, int width, int height)
{
  for (int j = x; j < x + width; j++)
    for (int i = y; i < y + height; i++)
      cur_img (i, j) = compute_color (i, j);

  return 0;
}
```

Exemple

```c
// Tile computation
int invert_do_tile_default (int x, int y, int width, int height)
{
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);

  return 0;
}

int invert_do_tile_bad (int x, int y, int width, int height)
{
  for (int j = x; j < x + width; j++)
    for (int i = y; i < y + height; i++)
      cur_img (i, j) = compute_color (i, j);

  return 0;
}
```

```
┌──(~/easypap)
└─> ./run -k invert  -i 10  -l images/firenze.png -n
Using kernel [invert], variant [seq], tiling [default]
Computation completed after 10 iterations
57.620
┌──(~/easypap)
└─> ./run -k invert  -i 10  -l images/firenze.png -n -wt bad
Using kernel [invert], variant [seq], tiling [bad]
Computation completed after 10 iterations
2617.541
```

X 45

Localité spatiale

- Forte probabilité qu'une donnée soit accédée prochainement…
  - Si elle est à proximité d'une donnée accédée récemment
    - Éléments de tableaux, champs de structures, etc.
- Les caches mémorisent des « lignes » (typiquement 64 octets)
  - Sur une architecture 64bits, une ligne de cache de 64 octets contient 8 mots
    - 8 double / long
    - 16 float / int

Cache

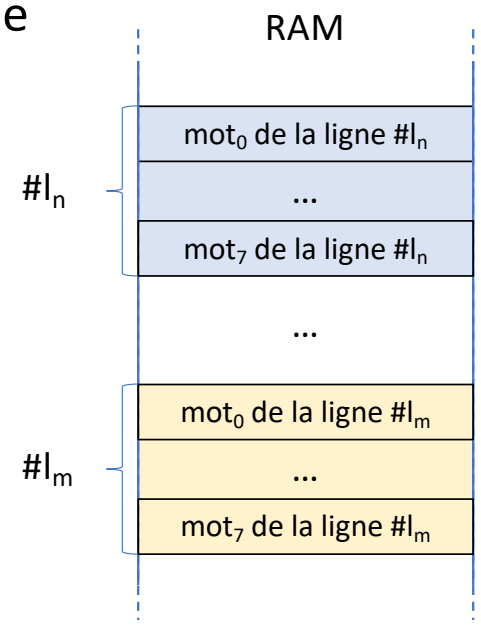
($\#l_n$, [mot_0, mot_1, …, mot_7], clean/dirty)
($\#l_m$, [mot_0, mot_1, …, mot_7], clean/dirty)
-
-
-
-

La mémoire est une séquence de lignes

- La RAM ne travaille qu'avec des lignes
  - Lorsque le CPU réclame un mot absent du cache, c'est toute la ligne de cache qui le contient qui est chargée depuis la RAM vers le cache

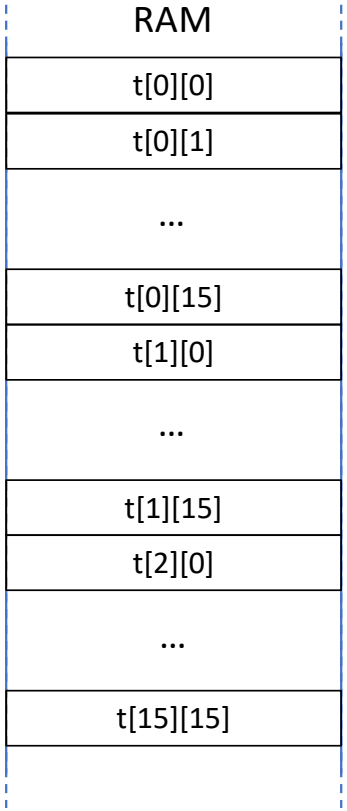
Cache

($\#l_n$, [mot_0, mot_1, …, mot_7], clean/dirty)
($\#l_m$, [mot_0, mot_1, …, mot_7], clean/dirty)
-
-
-
-



De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire suivant les lignes
  - Exemple :
    float t [16][16];
  - t occupe 16 x 16 x 4 octets (1 KB)



De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes
  - j=0, i=0..7 : le cache est rempli !

Cache

#l_n:	t[0][0], t[0][1], …, t[0][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

  - Avec un cache de 8 lignes
  - j=0, i=8 : première éviction ☹

Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+1}:	t[1][0], t[1][1], …, t[1][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += tab[i][j];
```

  - Avec un cache de 8 lignes
  - J=0, i=9 : seconde éviction ☹☹

Cache

#l_{n+8}:	t[8][0], t[8][1], …, t[8][15]
#l_{n+9}:	t[9][0], t[9][1], …, t[9][15]
#l_{n+2}:	t[2][0], t[2][1], …, t[2][15]
#l_{n+3}:	t[3][0], t[3][1], …, t[3][15]
#l_{n+4}:	t[4][0], t[4][1], …, t[4][15]
#l_{n+5}:	t[5][0], t[5][1], …, t[5][15]
#l_{n+6}:	t[6][0], t[6][1], …, t[6][15]
#l_{n+7}:	t[7][0], t[7][1], …, t[7][15]

De l'importance de l'agencement des données

- Les tableaux C sont linéarisés en mémoire de façon « *row-major* »
  - Exemple :

```
float t [16][16];

for (int j = 0; j < 16; j++)
  for (int i = 0; i < 16; i++)
    sum += t[i][j];
```

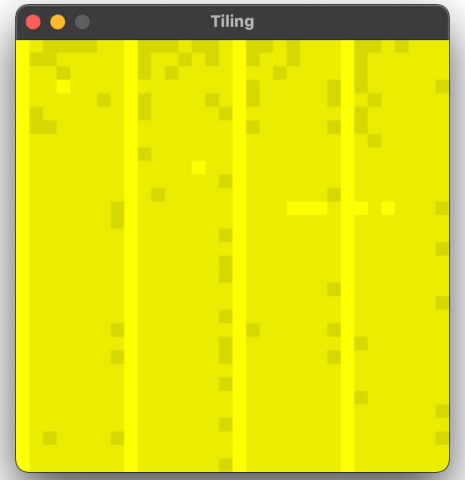
  - En fait, chaque accès va désormais provoquer une éviction…
  - Si possible, il faut parcourir t en ligne
  - Localité spatiale : prefetching matériel (limite = page)

Cache

$\#l_{n+8}$:	t[8][0], t[8][1], …, t[8][15]
$\#l_{n+9}$:	t[9][0], t[9][1], …, t[9][15]
$\#l_{n+2}$:	t[2][0], t[2][1], …, t[2][15]
$\#l_{n+3}$:	t[3][0], t[3][1], …, t[3][15]
$\#l_{n+4}$:	t[4][0], t[4][1], …, t[4][15]
$\#l_{n+5}$:	t[5][0], t[5][1], …, t[5][15]
$\#l_{n+6}$:	t[6][0], t[6][1], …, t[6][15]
$\#l_{n+7}$:	t[7][0], t[7][1], …, t[7][15]

Exemple : kernel invert d'easypap

- Influence de la taille de la tuile
- Prefetching matériel (limite = page) / logiciel



Prefetching logiciel

```
// Tile computation
int invert_do_tile_prefetch (int x, int y, int width, int height)
{
  volatile long tmp[DIM / 1024 + 2] __attribute__ ((unused));
  for (int i = y; i < y + height; i++) {
    for (int k = 0; k < width; k += 1024) {
      tmp[k / 1024] = cur_img (i, x + k);
    }

    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
  }
  return 0;
}
```

Réalisation matérielle des caches

- Une petite mémoire rapide proche du pipeline.
- Capable de conserver des données récemment accédées.
- Capable de dire si elle détient ou non une donnée d'après son adresse.

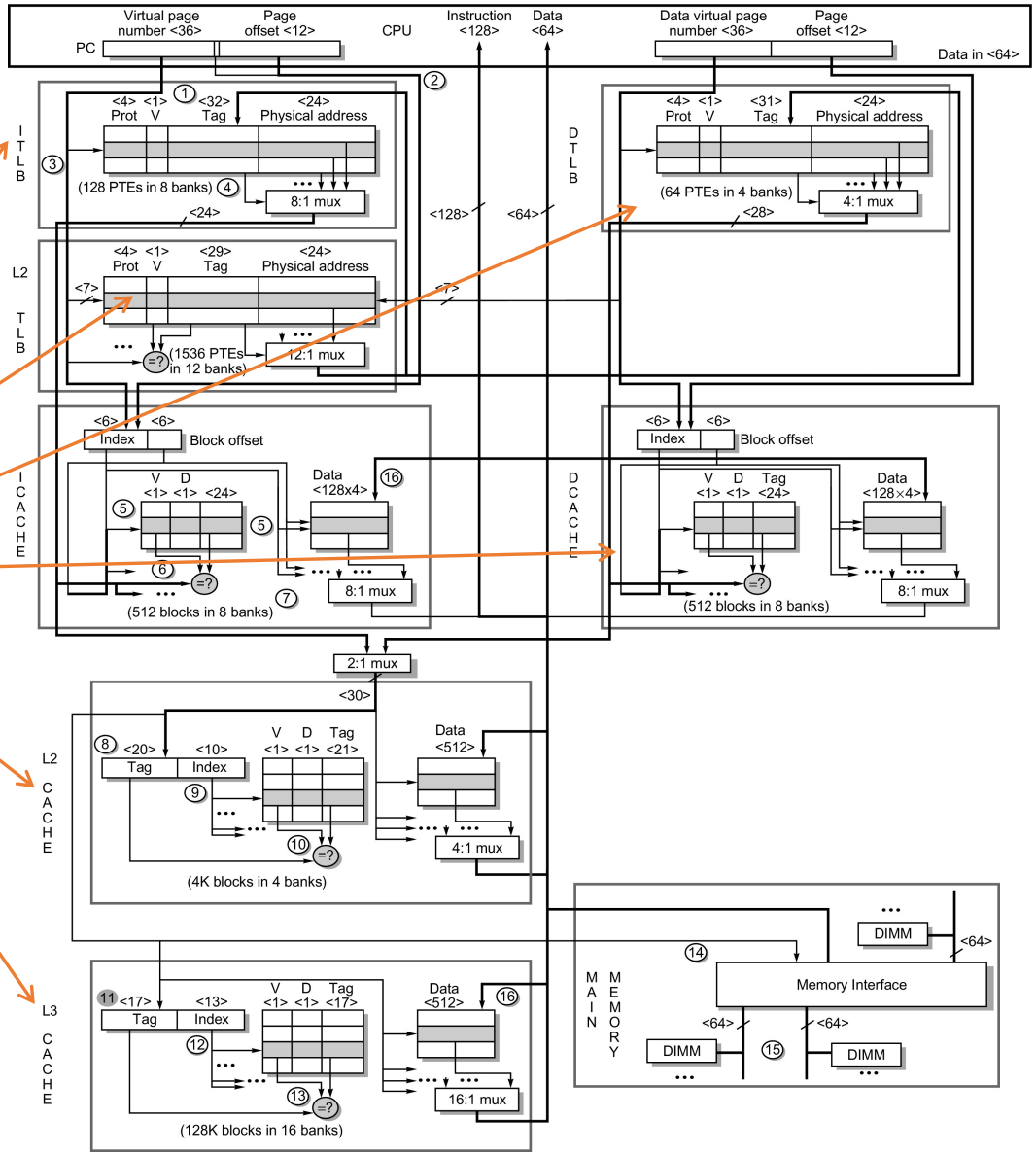
Un cache est une table de hachage réalisée matériellement :

- La clef est l'adresse, ie. la fonction de hachage = f(adresse)
- Le nombre de conflits possibles est limité car gravé en silicium

- La fonction de hachage doit optimiser la fréquence des conflits
  - Tenir compte de la règle de localité spatiale et temporelle

Tables de hachage

Cache du Core I7



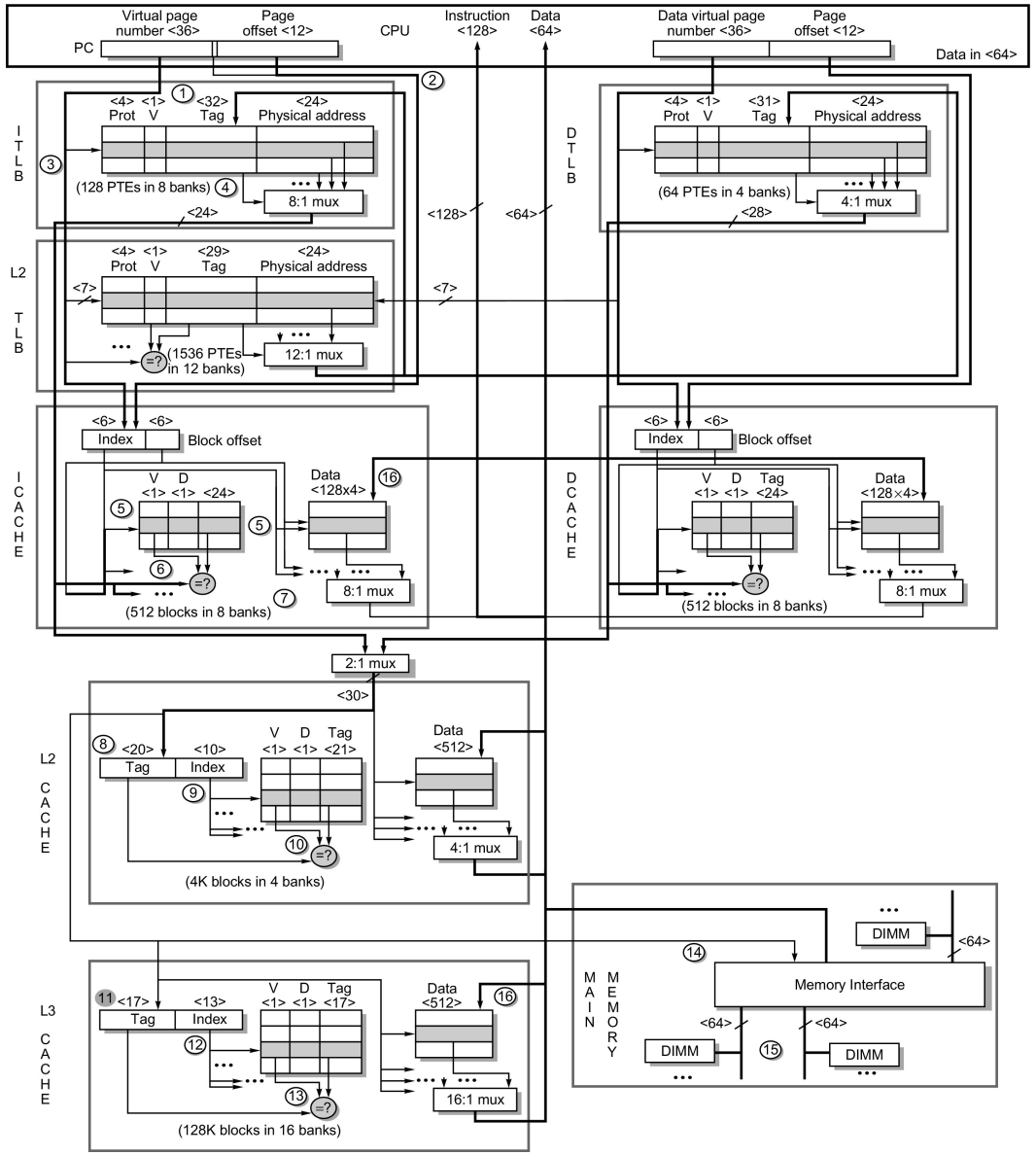
TLB L1
instructions

TLB L2
I & D

Cache L1
Instructions

Cache L2
I & D

Cache L3
I & D



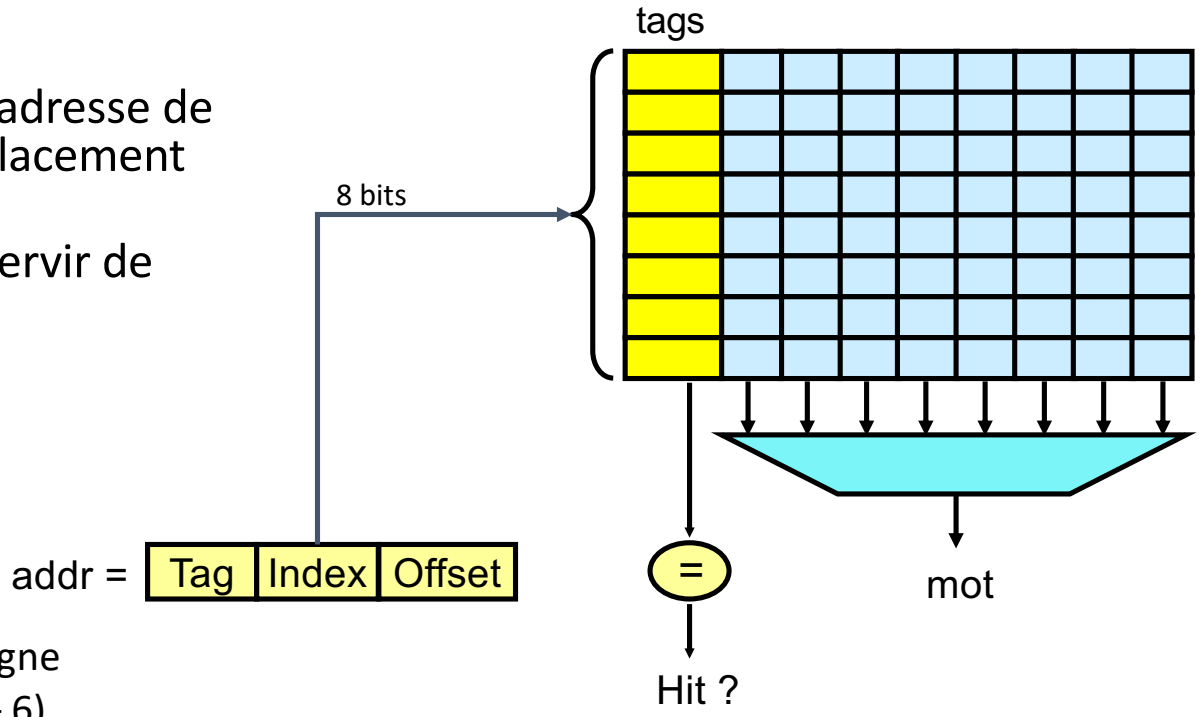
TLB L1
données

Cache L1
Données

Mémoire

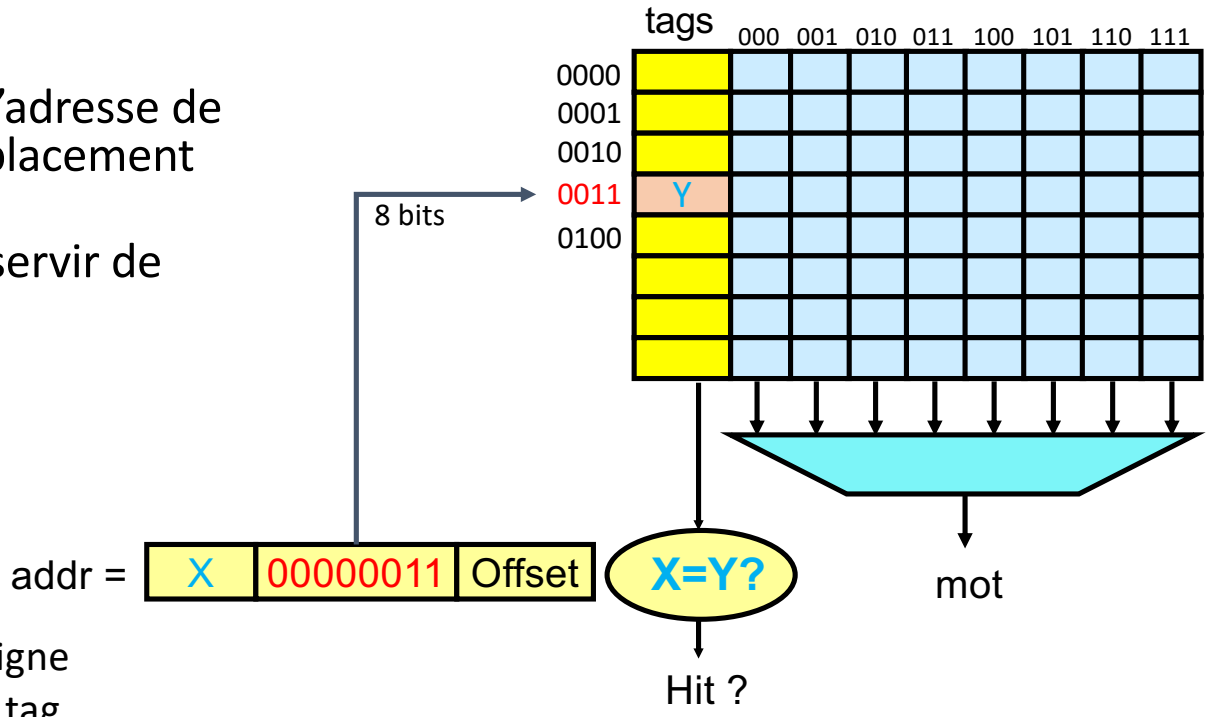
Direct-Mapped Cache

- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (table de hachage)
- Exemple
  - Cache de 16 KB
    - 256 x 64 octets
    - Index sur 8 bits pour déterminer le numéro de ligne
    - 22 bits pour le tag (36 – 8 – 6)



Direct-Mapped Cache

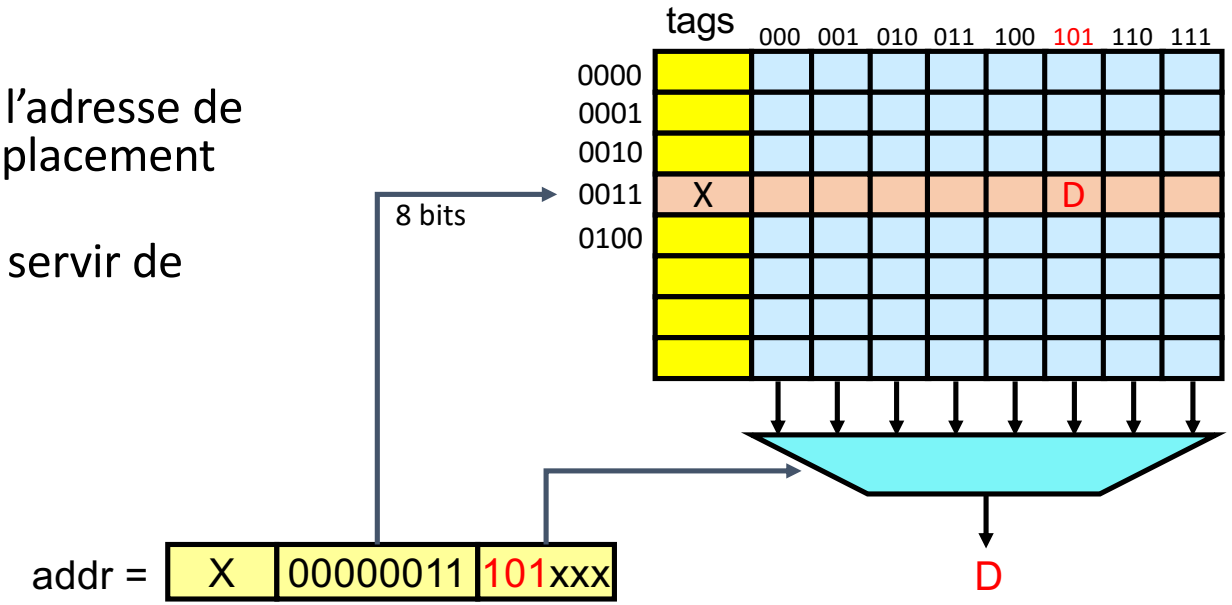
- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (table de hachage)
- Exemple
  - Cache de 16 KB
    - 256 x 64 octets
    - Index sur 8 bits pour déterminer le numéro de ligne
    - 36 – 6 – 8 = 22 bits pour le tag



Direct-Mapped Cache

- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (table de hachage)

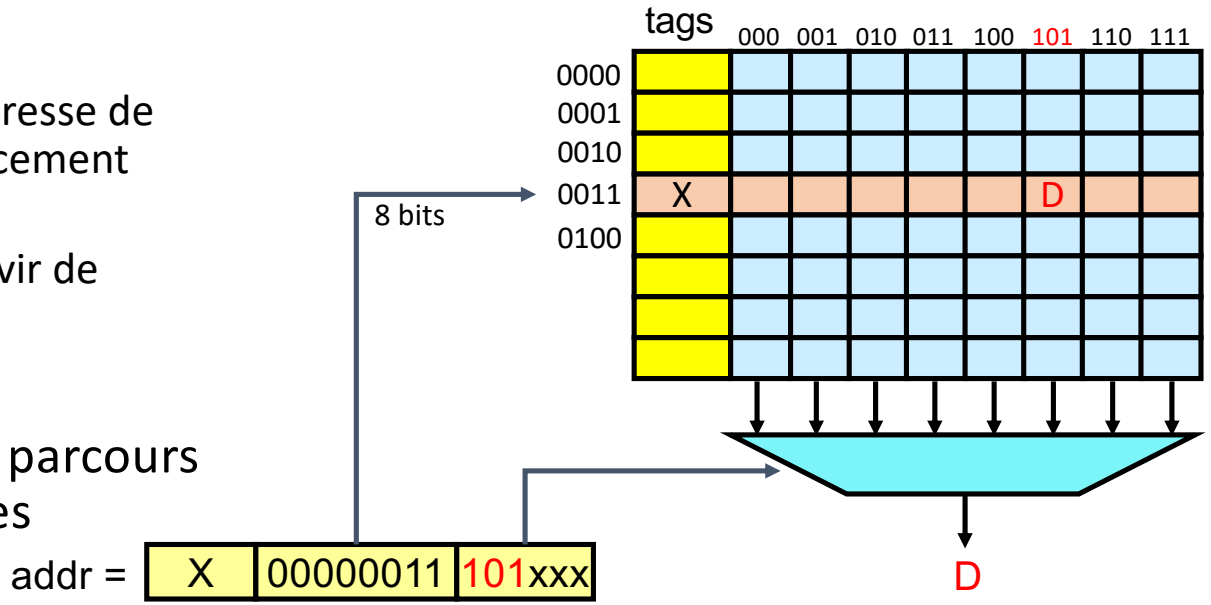
- Exemple
  - Cache de 16 KB
    - 256 x 64 octets
    - Index sur 8 bits pour déterminer le numéro de ligne
    - 22 bits pour le tag (36 – 8 – 6)



Direct-Mapped Cache

- Idée
  - Les bits de poids faible de l'adresse de ligne prédéterminent l'emplacement dans le cache
  - Les bits de poids fort vont servir de « tag » (table de hachage)

- Parfaitement adapté pour le parcours d'un tableau suivant les lignes



Exemple : i9-13900K

CPU Name	13th Gen Intel® Core™ i9-13900K
Threading	1 CPU - 8 Core - 16 Threads
Frequency	3000 MHz (30 * 100 MHz) - Uncore: 2000 MHz
Multiplier	Current: 30 / Min: 8 / Max: 55
Architecture	Raptor Lake / Stepping: B0 / Technology: 10 nm
L1 Data Cache	8 x 48 KB (12-way, 64-byte line)
L1 Inst. Cache	8 x 32 KB (8-way, 64-byte line)
L2 Cache	8 x 2 MB (16-way, 64-byte line)
L3 Cache	36 MB (12-way, 64-byte line)

Comment optimiser l'utilisation du cache

- Types de défauts de cache (*cache misses*)
  - ***Cold miss*** : premier accès à une variable
    - Remède : regrouper des variables (structures, tableaux)
  - ***Capacity miss :*** le cache est complet
    - Revoir l'ordre des boucles, revoir l'algorithme
    - Travailler plus intensivement sur une plus petite zone de données (cf tuile)
  - ***Conflict miss*** : le cache n'est pas complet mais son associativité est trop faible.
    - Regrouper des variables (structures, tableaux)
    - Revoir l'alignement des données

Exemple : le programme film

- Séquence de N images 1024 x 1280 pixels (32 bits)
- On cherche à calculer le nombre de pixels communs aux images consécutives

QCM Quel est le code le plus rapide ?

A

```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

B

```
for(int i=0; i < 1024; i++)
  for(int n=0; n < N-1; n++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

C

```
for(int i=0; i < 1024; i++)
   for(int j=0; j < 1280; j++)
     for(int n=0; n < N-1; n++)
       if (film[n][i][j] != film[n+1][i][j])
         diff[n]++;
```

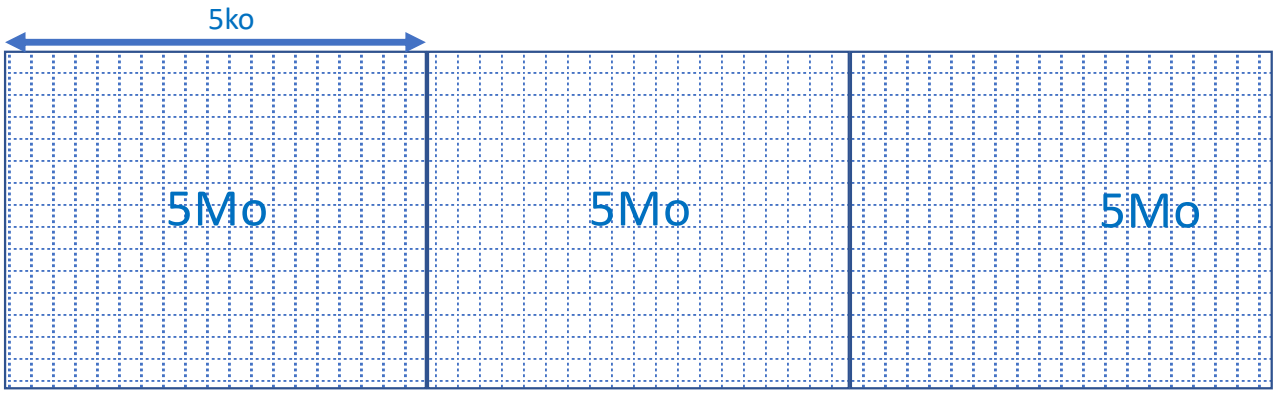
D

```
for(int j=0; j < 1280; j++)
  for(int i=0; i < 1024; i++)
     for(int n=0; n < N-1; n++)
       if (film[n][i][j] != film[n+1][i][j])
         diff[n]++;
```

QCM Quel est le code le plus rapide ?

A

```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```



Cache L3 de 8Mo

QCM Quel est le code le plus rapide ?

A

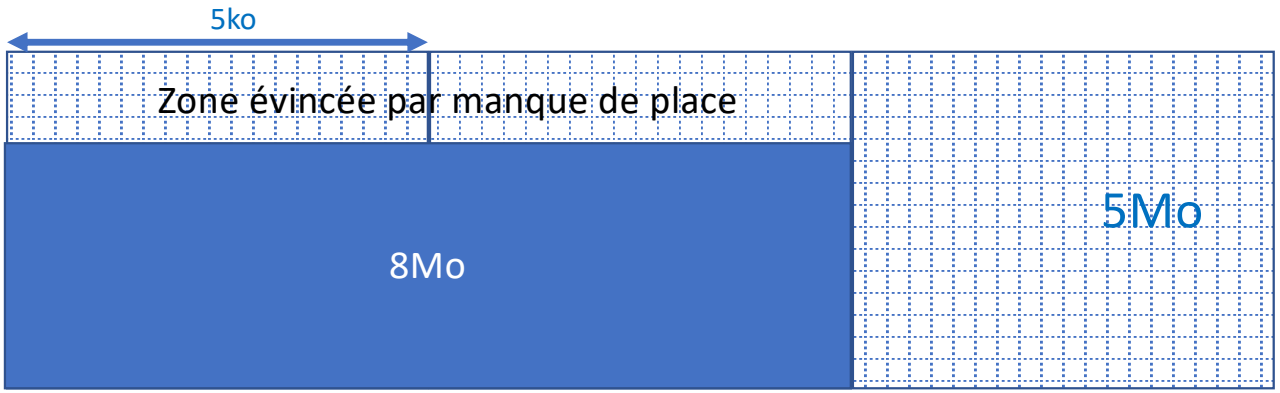
```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

Cache L3 de 8Mo

QCM Quel est le code le plus rapide ?

A

```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

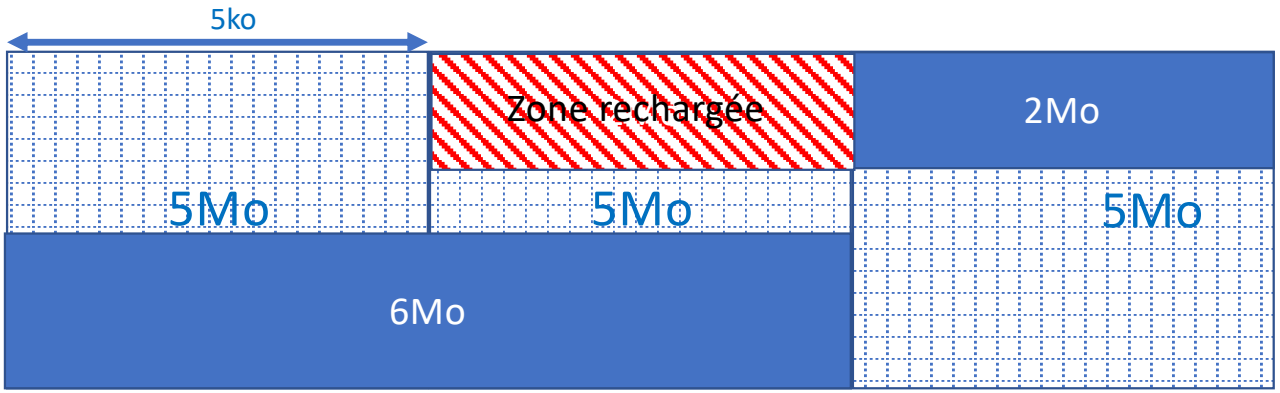


Cache L3 de 8Mo

QCM Quel est le code le plus rapide ?

A

```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

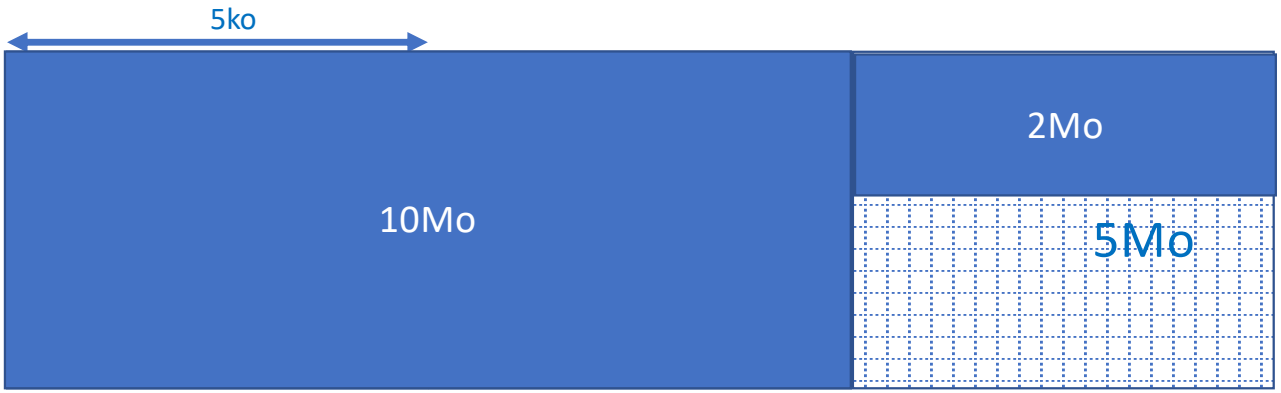


Chaque pixel des images internes est chargé deux fois dans le cache L3.

QCM Quel est le code le plus rapide ?

A

```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```



Cache de L3 12Mo : ok

QCM Quel est le code le plus rapide ?

A

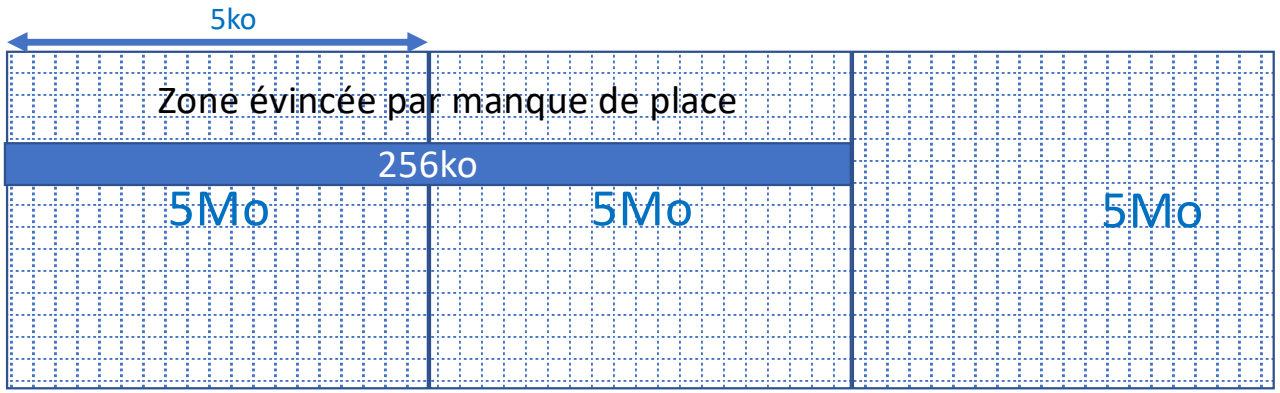
```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

Cache de 12Mo

QCM Quel est le code le plus rapide ?

A

```
for(int n=0; n < N-1; n++)
  for(int i=0; i < 1024; i++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

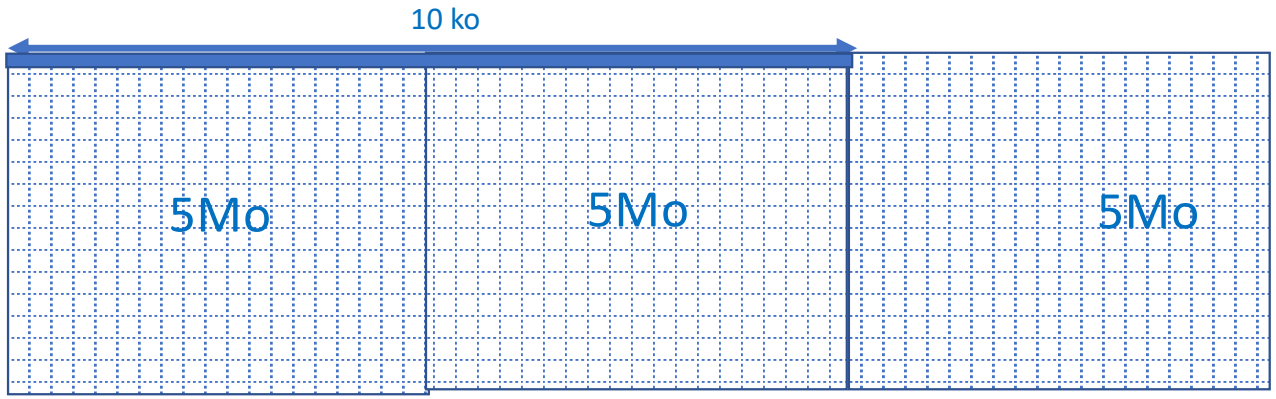


Cache L2 de 256ko

QCM Quel est le code le plus rapide ?

B

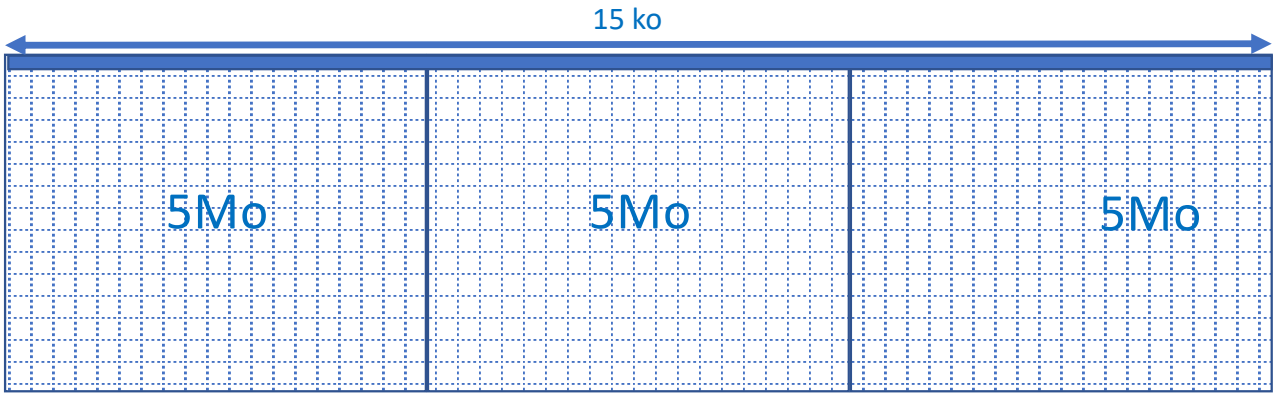
```
for(int i=0; i < 1024; i++)
  for(int n=0; n < N-1; n++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```



QCM Quel est le code le plus rapide ?

B

```
for(int i=0; i < 1024; i++)
  for(int n=0; n < N-1; n++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```

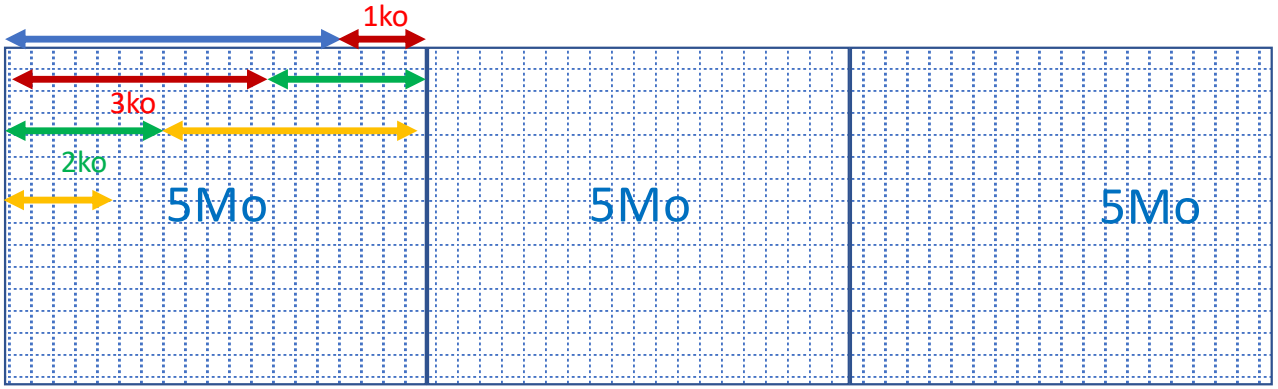


15 ko tiennent dans le cache L1 les pixels pourraient être chargés qu'une fois
Si l'associativité est bonne.

QCM Quel est le code le plus rapide ?

B

```
for(int i=0; i < 1024; i++)
  for(int n=0; n < N-1; n++)
    for(int j=0; j < 1280; j++)
      if (film[n][i][j] != film[n+1][i][j])
        diff[n]++;
```



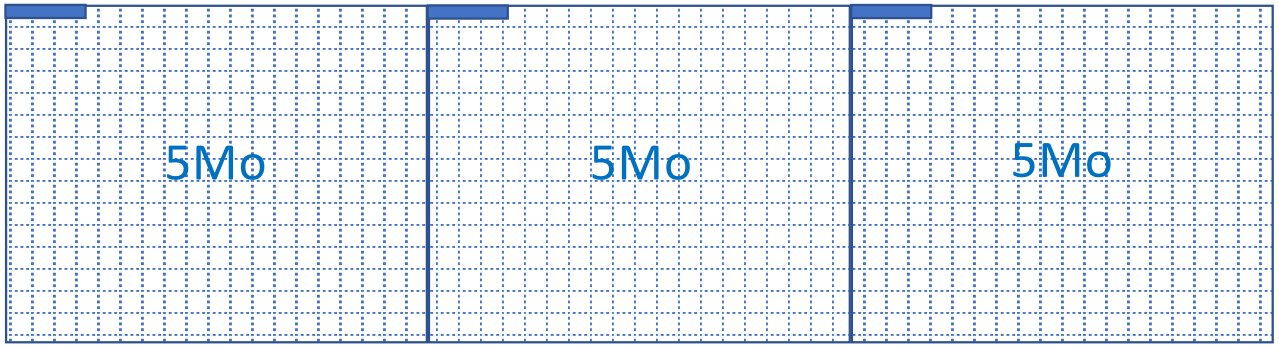
Les pages ne sont pas consommées en entier ➔ le prefetch matériel n'est pas optimal.

QCM Quel est le code le plus rapide ?

C

```
for(int i=0; i < 1024; i++)
   for(int j=0; j < 1280; j++)
     for(int n=0; n < N-1; n++)
       if (film[n][i][j] != film[n+1][i][j])
         diff[n]++;
```

64 octets



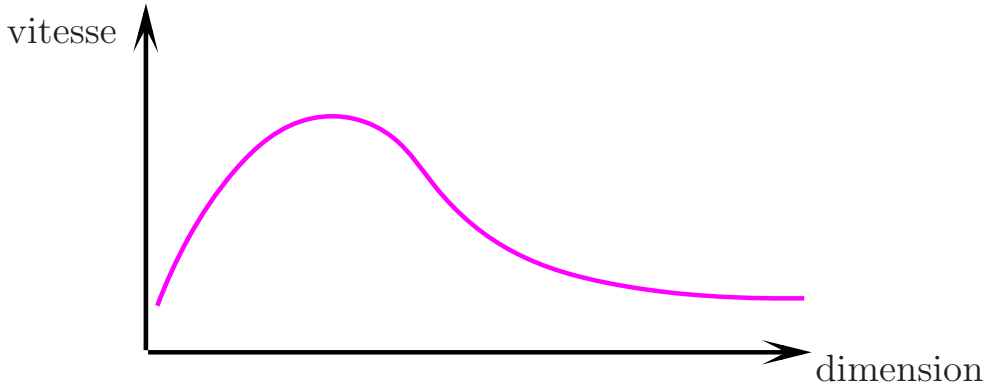
Dangereux à cause de l'associativité.

Optimisation de l'utilisation du cache

- Minimiser le nombre de défauts de cache
  - Réutiliser les données chargées dans le cache
    - Adapter les structures de données
    - Aligner les données

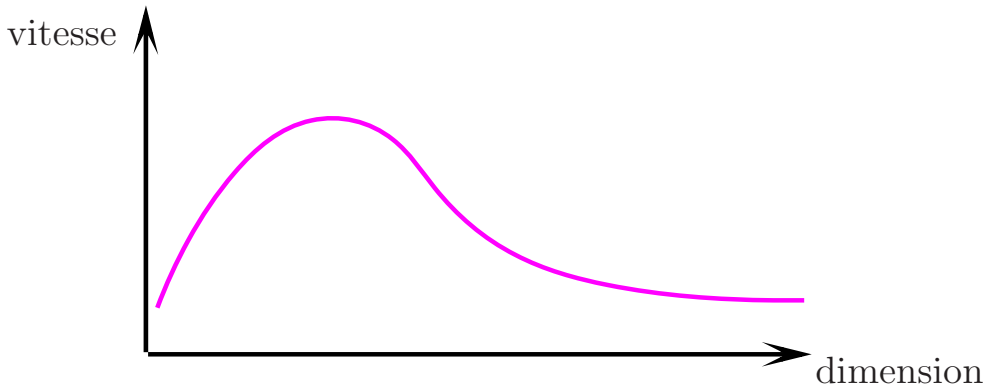
- Techniques de pavage (tiling)

```
for( I =; …; I+=TI)
    for( J =; …; J+=TJ)
        for( i = I; i < I + TI ; i++)
            for( j = J; j < J + TJ ; j++)
```



Technique de pavage omp 5.1

```
void func1(int A[100][128]) {
#pragma omp parallel for
#pragma omp tile sizes(5,16)
for (int i = 0; i < 100; ++i)
   for (int j = 0; j < 128; ++j)
     A[i][j] = i*1000 + j;
}
void func2(int A[100][128]) {
#pragma omp parallel for
for (int i1 = 0; i1 < 100; i1+=5)
  for (int j1 = 0; j1 < 128; j1+=16)
    for (int i2 = i1; i2 < i1+5; ++i2)
      for (int j2 = j1; j2 < j1+16; ++j2)
        A[i2][j2] = i2*1000 + j2;
}
```

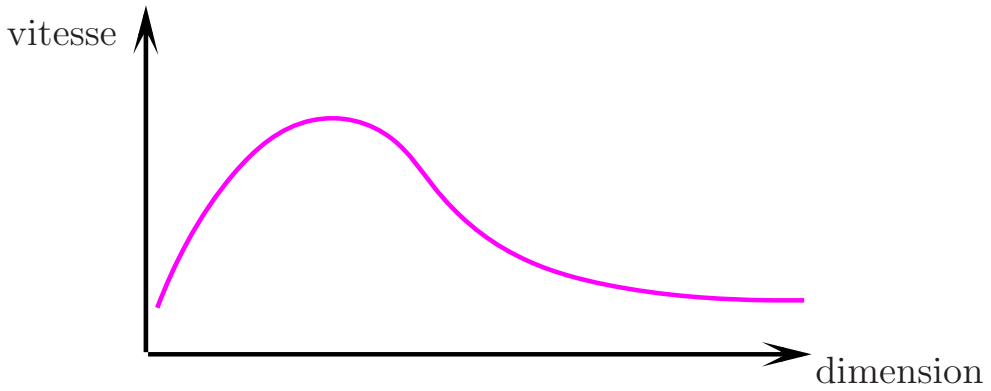


Optimisation de l'utilisation du cache

- Minimiser le nombre de défauts de cache
  - Cache oblivious (Cilk sort)
  - Cache conscious https://rrze-hpc.github.io/layer-condition/#calculator

- Techniques de pavage (tiling)

```
for( I =; …; I+=TI)
    for( J =; …; J+=TJ)
        for( i = I; i < I + TI ; i++)
            for( j = J; j < J + TJ ; j++)
```



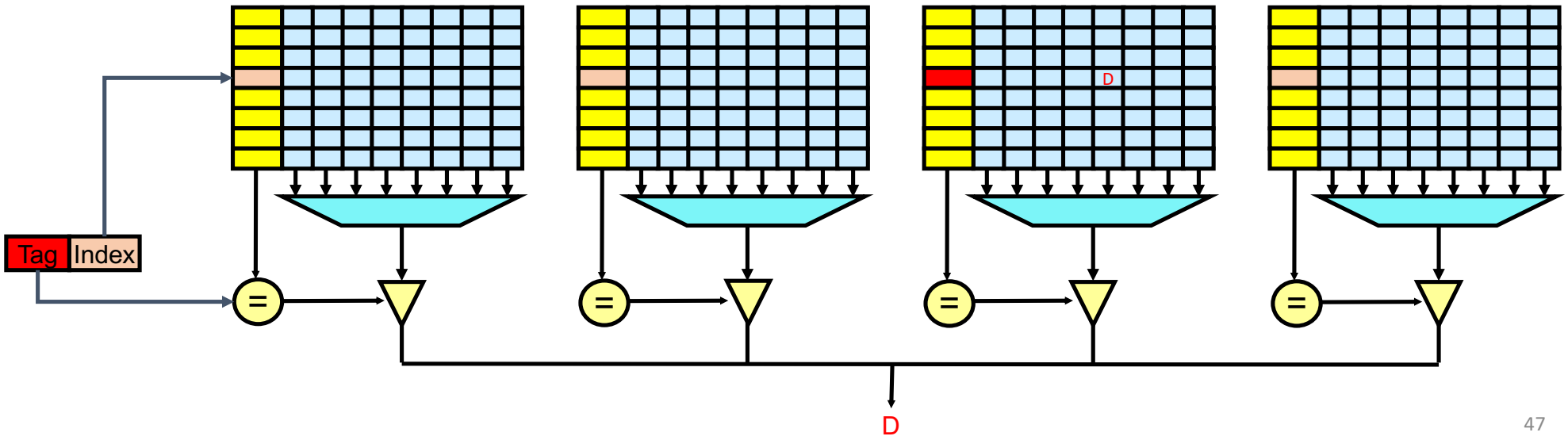
Nachos – Devoir 4

- Programmes OpenMP en mode utilisateur !
- Bonus
  - Exploitation des cartes graphiques

- Date de rendu
  - Fin de semaine prochaine

Back to N-way associative Caches

- On utilise N sous-caches directs (= bancs)
  - N lignes de même index (mais de tag différent) peuvent co-exister
  - Exemple: 4 voies x 128 lignes x 64 octets = 4 bancs de 8 BKB = 32 KB



N-way associative Cache

- Exemple : core i7
  - Cache L1D du Intel core i7
    - 32KB, 8 voies, 64 lignes de 64 octets par voie
    - 8 bancs de 4 KB
    - Index = 6 bits
  - Cache L2
    - 256KB, 8 voies, 512 lignes de 64 octets par voie
    - 8 bancs de 32 KB
    - Index = 9 bits
  - Cache L3
    - 8MB, 16 voies, 8192 lignes de 64 octets par voie
    - 16 bancs de 512 KB
    - Index = 13 bits

N-way associative Cache

- Exemple : core i7
  - Cache L1D du Intel core i7
    - 32KB, 8 voies, 64 lignes de 64 octets par voie
    - 8 bancs de 4 KB

    - Si la variable x est dans le cache, alors
      - (&x + 1*4096),
      - (&x + 2*4096)
      - …
      - (&x + 7*4096) pourront entrer dans le cache, chacune dans un banc différent

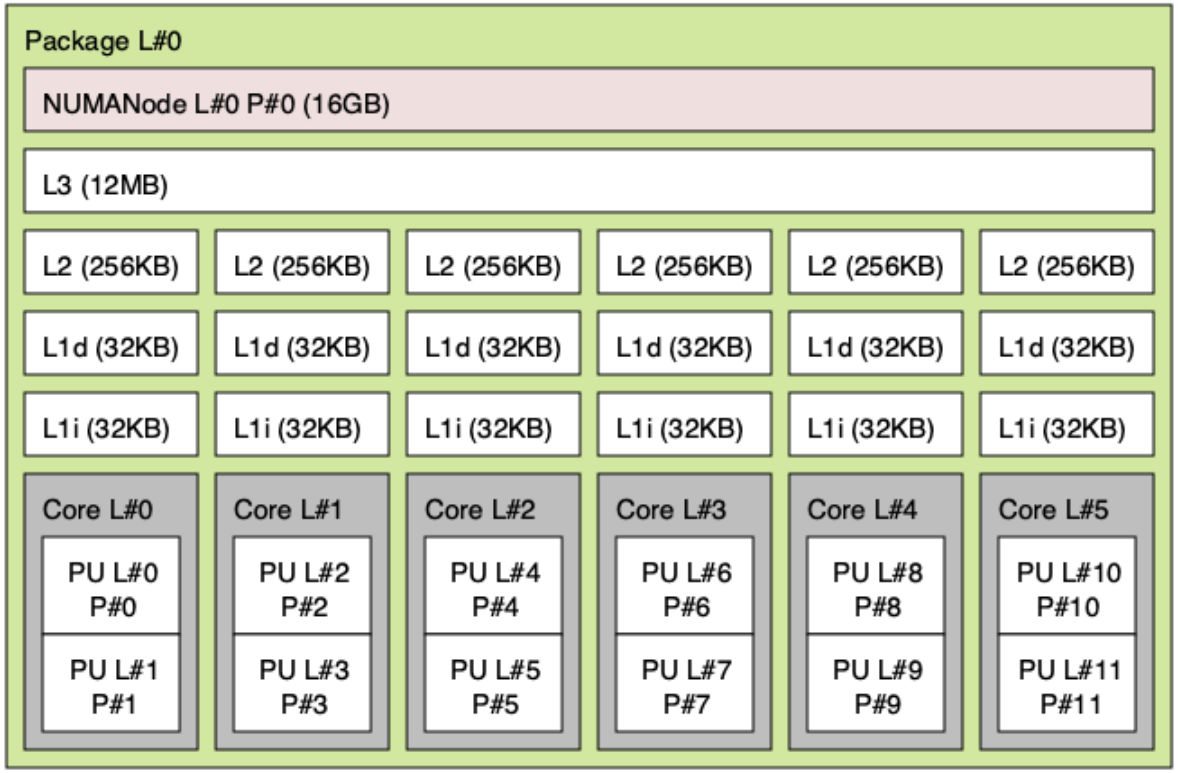
      - Un accès à (&x + 8*4096) provoquera l'éviction d'une des huit lignes
      - Attention aux strides multiples de 4KB !

N-way associative Cache

- Finalement, les conflits surgissent moins vite avec une associativité élevée
  - Exemple : 16 voies pour de nombreux caches L3

- Pourquoi ne pas augmenter franchement l'associativité et proposer 128 ou 1024 voies ?
  - Parce qu'en augmentant le nombre de bancs et en réduisant le nombre de lignes par banc, on augmente le nombre de comparateurs et la complexité des multiplexeurs
  - À l'extrême, on tendrait vers un cache *full associative* !

Caches et multicœurs



Objectifs

- Le calcul doit être juste
  - Protocole MESI
- Faire en sorte que les cœurs coopèrent sans se gêner
  - Les accès à la mémoire doivent être optimisés
    - Les caches L1 sont privés, les L3 sont partagés
    - Les données ne doivent pas voyager de cache en cache
    - OMP_SCHEDULE static vs dynamic vs nonmonotonic:dynamic

Caches et multicœurs

- La mémoire et les caches forment une hiérarchie
  - Très variable d'une architecture/d'un constructeur à l'autre
- Certains caches sont privés, d'autres partagés par un sous-ensemble (éventuellement tous) les coeurs
- Il faut garantir la cohérence des données en présence de copies !

Synchronisation de la mémoire

Consistance séquentielle (Lamport 1979)

→ *Sémantique de l'entrelacement* :

- Le résultat de l'exécution de $a_1 a_2 \ldots a_q$ // $b_1 b_2 \ldots b_p$ doit correspondre à un programme séquentiel où les a_i sont exécutés dans l'ordre ainsi que les b_i.

- Exemple : au départ { x = 0, y = 0} on lance 2 threads :

  T1 : { x = 1 ; print(y); }

  T2 : { y = 1; print(x); }

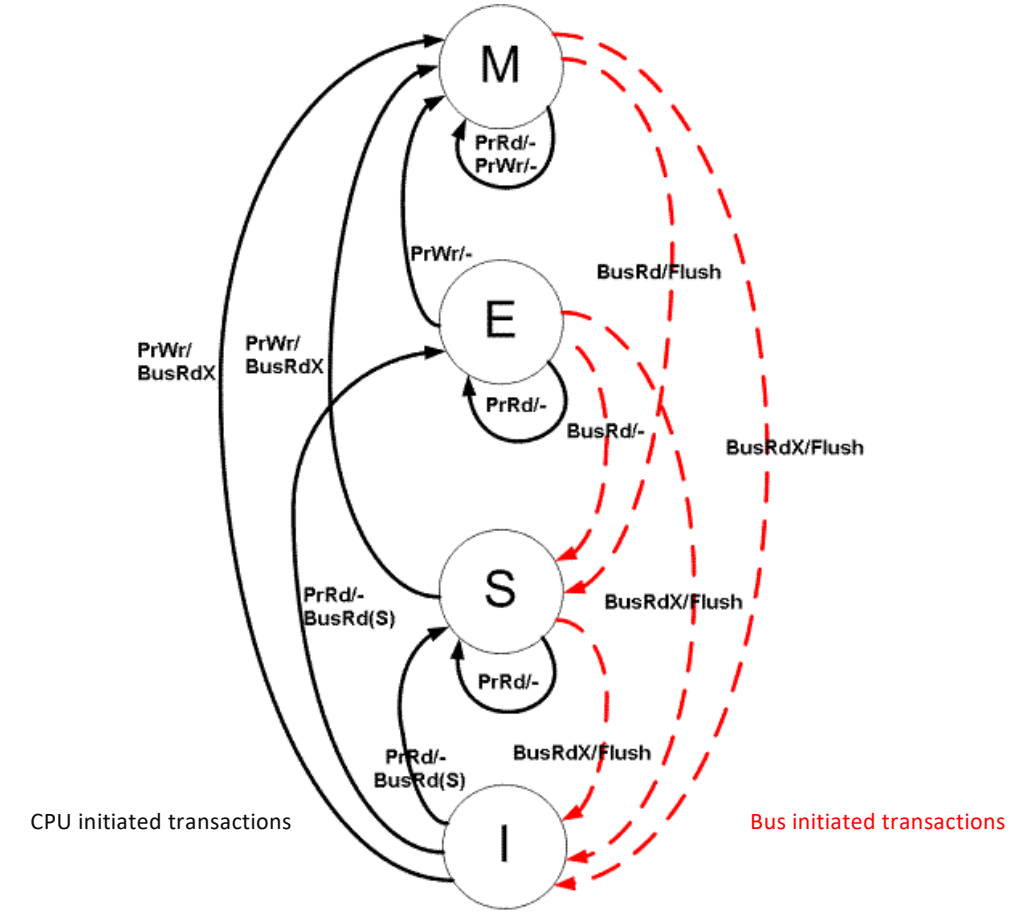
L'affichage ne peut pas être 0 0 - autrement on est bigrement étonné !

Caches et multicœurs

- Il faut garantir la cohérence des données en présence de copies !
  - Une donnée peut être en mémoire, dans un ou plusieurs caches, dans un registre.
- La cohérence registre / cache est à la charge du programmeur
  - Instructions atomiques
- Protocole de cohérence de cache implémenté matériellement
  - Assurer la cohérence et optimiser les échanges entre les caches et la mémoire
  - Comment :
    - Un état est associé à chaque ligne du cache
    - Différents protocole : MSI, MESI, MOESI, MESIF, etc.

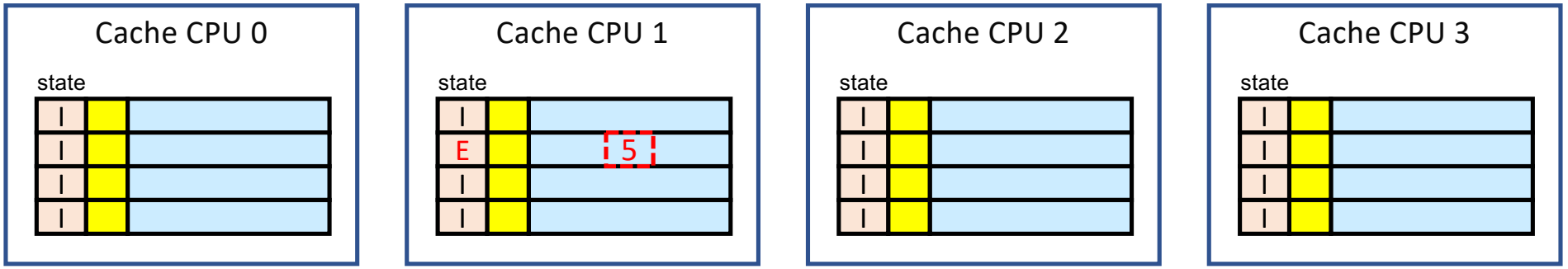
Caches et multicœurs

- Le protocole MESI
  - Chaque ligne de cache peut être dans l'un de ces 4 états :
    - M (modified)
      - La ligne n'est présente que dans le cache actuel, et est *dirty*
    - E (exclusive)
      - La ligne n'est présente que dans le cache actuel, et est *clean*
    - S (shared)
      - La ligne est peut-être présente dans d'autres caches, et est *clean*
    - I (invalid)
      - La ligne ne contient pas de données



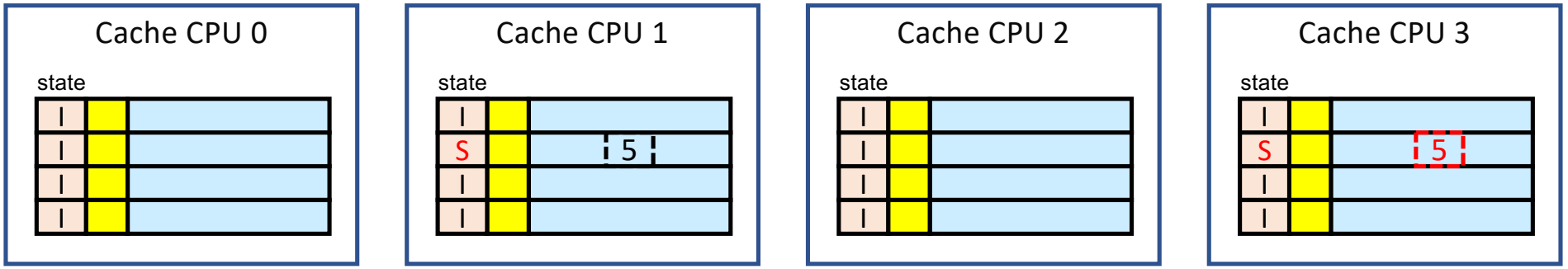
Caches et multicœurs

- CPU_1: PrRd (x) (qui vaut 5 en RAM)
  - Le signal BusRd est envoyé sur le bus
  - Les autres caches répondent qu'ils ne possèdent pas de copie
  - La ligne de cache vient de la RAM, et son état est *Exclusive*



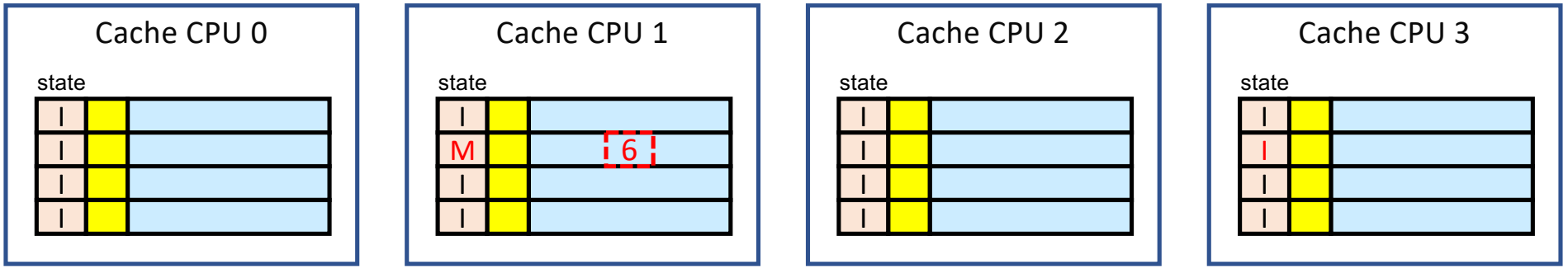
Caches et multicœurs

- CPU_3: PrRd (x)
  - Le signal BusRd est envoyé sur le bus
  - CPU_1 envoie la ligne de cache à CPU_3
  - L'état de la ligne passe à *Shared* pour CPU_1 et CPU_3



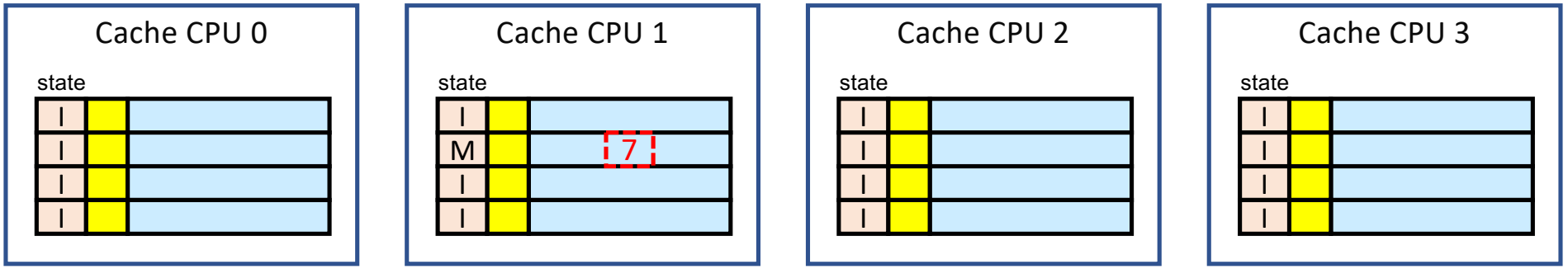
Caches et multicœurs

- CPU_1: PrWr (x, 6)
  - Le signal BusRdX est envoyé sur le bus
  - CPU_3 invalide sa copie
  - L'état de la ligne passe à *Modified* pour CPU_1



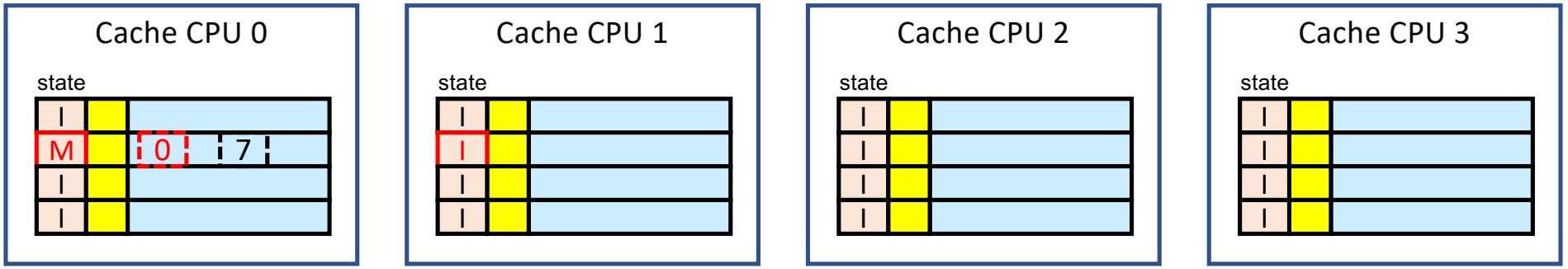
Caches et multicœurs

- CPU_1: PrWr (x, 7)
  - Aucun signal n'est envoyé sur le bus
  - L'état de la ligne reste à *Modified*



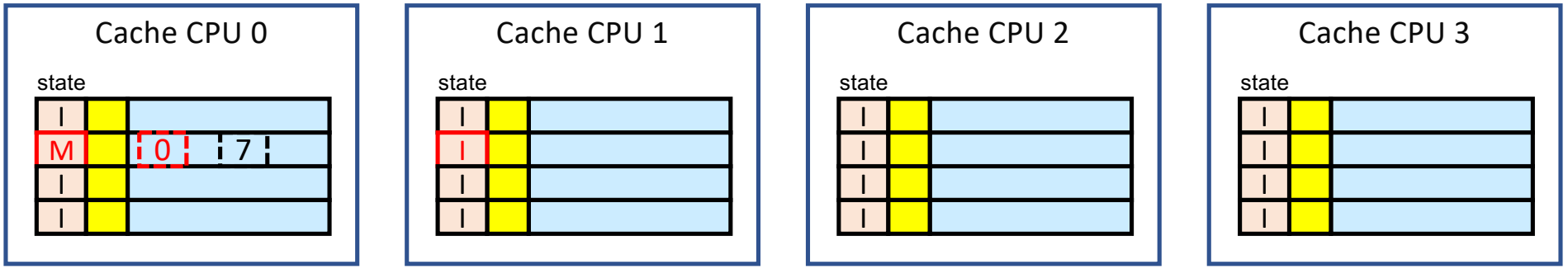
Caches et multicœurs

- CPU_0: PrWr (y, 0), y se trouvant dans la même ligne de cache que x
  - Le signal BusRdX est envoyé sur le bus
  - CPU_1 envoie la ligne à CPU_0 puis invalide sa copie
  - L'état de la ligne passe à *Modified* pour CPU_0



Caches et multicœurs

- Si CPU_1 modifie x, puis CPU_0 modifie y, etc.
  - On assiste à un ping-pong entre les caches
  - Il s'agit d'une situation de faux-partage !
    - Introduction de *padding* nécessaire



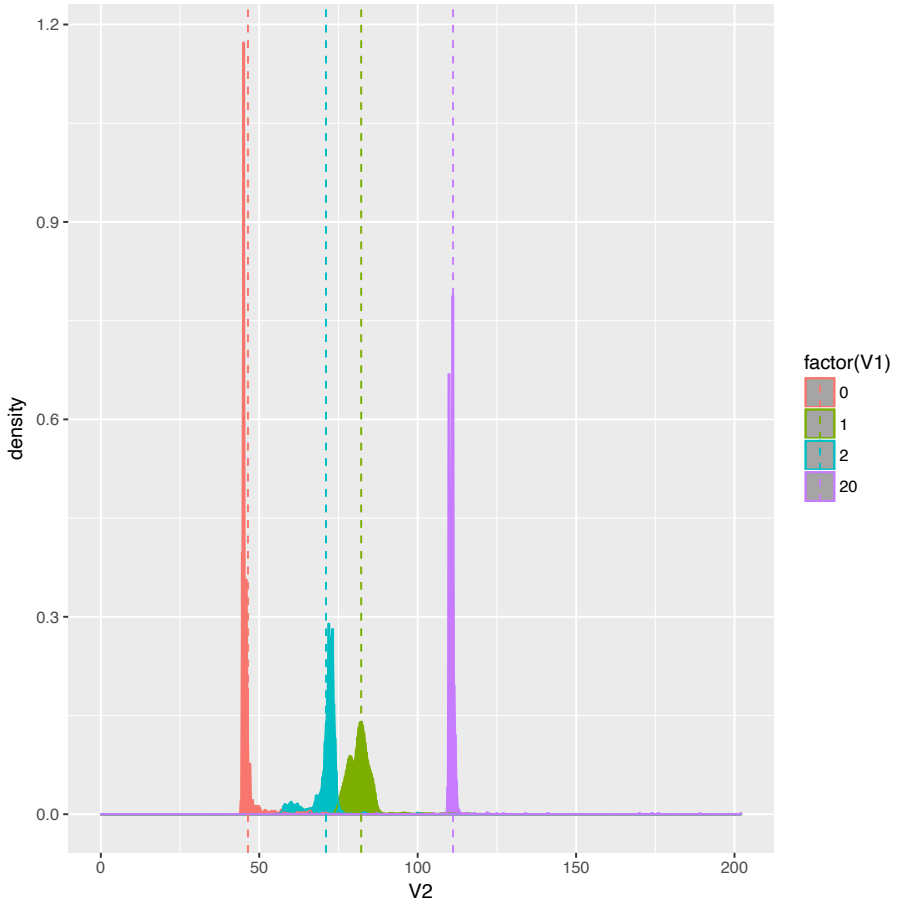
Caches et multicœurs

- Si CPU_1 modifie x, puis CPU_0 modifie y, etc.
  - On assiste à un ping-pong entre les caches
  - Il s'agit d'une situation de faux-partage !
    - Introduction de *padding* pour séparer les données

- À la lumière de ce constat, on peut se poser des questions sur l'efficacité d'un code tel que :

```
float tab [N];

#pragma omp parallel for schedule(static, 1)
      for (int i = 0; i < N; i++)
            tab [i] = f (i);
```

Coût du faux partage machine tesla




```
int Tab[N] ;

void * thread_fun(void *p)
{
        for(..)
        {
                Tab[id] +=  1;
                ...
```

Temps d'exécution de la boucle

46.439	sans partage
71.058	partage coeur voisin
82.171	partage entre 2 sockets
111.173	partage hyperthreading

Instructions atomiques

- Protéger une opération sur une *donnée unique* sans prendre de mutex
  - Assurer la cohérence séquentielle d'une donnée chargée dans un registre le temps d'un calcul élémentaire
  - Intuition : prolonger MESI jusque dans les registres

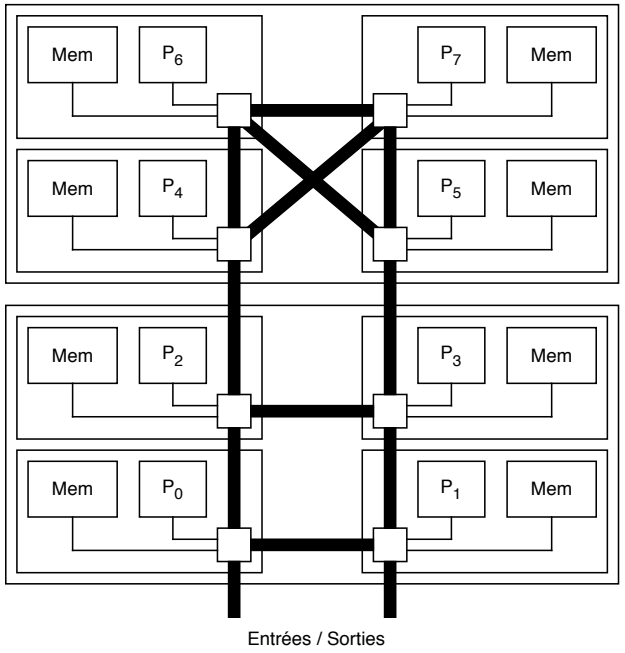
- Enchaînement *atomique* « lecture L1 - opération - écriture L1»
  - Réalisé via la *mémoire transactionnelle* :
    - Acquérir la ligne de cache et positionner son état à *modified ;*
    - Exécuter l'opération ;
    - Écrire le résultat que si la ligne cache est toujours là, recommencer autrement.
- builtin gcc

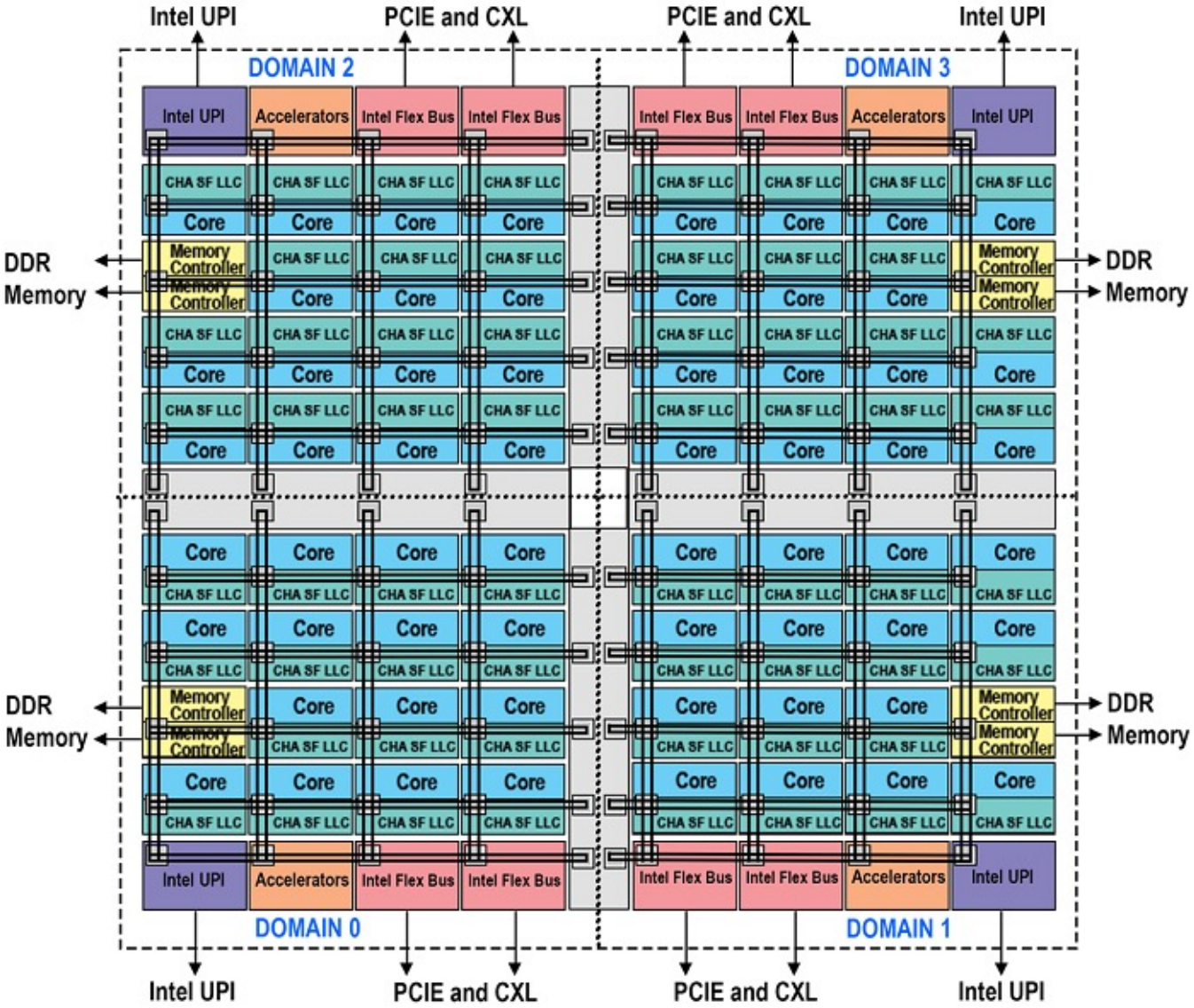
  type __sync_fetch_and_add (*type* *ptr, *type* value, ...)

  type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
- OpenMP
  - pragma omp capture

NUMA
non uniform memory access

- Répartition des bancs mémoire
  - Nœud numa : processeur + mémoire
  - SGI, IBM (90') - AMD, Intel (00')
- Réseau de communication inter nœud
  - Facteur NUMA : la latence mémoire dépend de la distance entre le processeur et la mémoire.
  - Parfois source de congestion !
- De nos jours tous les *gros* processeurs sont NUMA





Intel xeon

4th Gen (2022)

Sapphire Rapids

- 1 processeur = 4 noeuds
- Jusque 60 cœurs

Placement des threads et des données

Threads : ne pas laisser faire l'OS !

- Linux : pthread_attr_setaffinity_np()
- OpenMP
  - Variable OMP_PLACES
  - clause affinity() des tâches
  - Schedule : static ou nonmonotonic:dynamic (llvm)

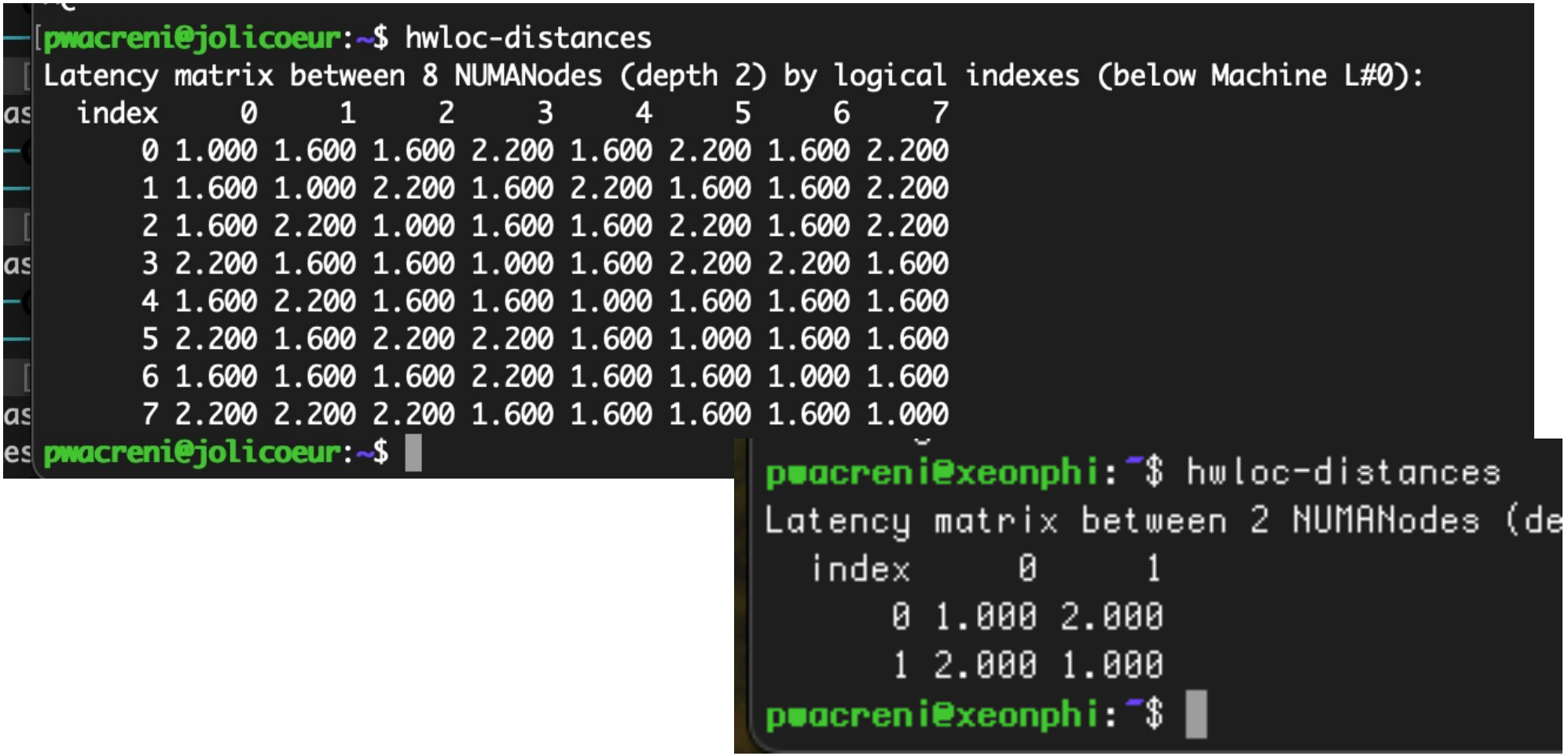
Données : se reposer sur l'OS !

- Allocation *first touch*
  - Allocation paresseuse des pages lors de la première utilisation
  - Par défaut la page est placée sur le nœud numa du cœur ayant causé l'allocation

Bibliothèques spécialisées

HWLoc, Likwid, Libnuma (linux)

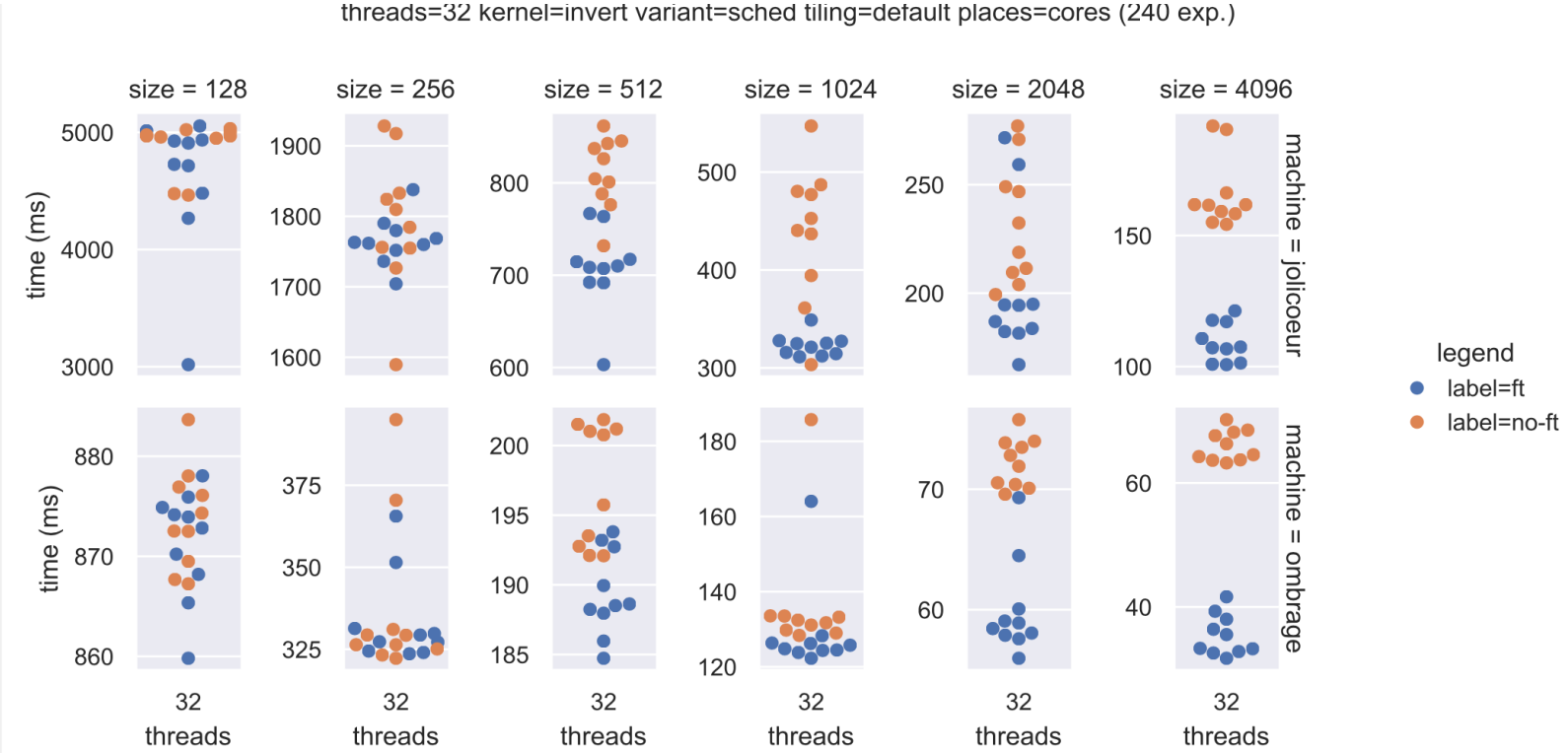
Serveur jolicoeur du cremi – 8 x 8 cœurs et xeonphi 2 x 10 coeurs



Kernel invert avec et sans stratégie first touch

AMD

Intel



plots/easyplot.py -if invert.csv --plottype catplot -x threads -y time --delete iterations tileh tilew schedule \
-- col=size row=machine sharey=false aspect=0.6 height=2

Placement OpenMP

- Variable **OMP_PLACES** pour décrire l'architecture
  - OMP_PLACES= '{cpu-id de la 1ere place}, {cpu-id de la 2ieme place},…'
- Variable **OMP_PROC_BIND**
- Clause **proc_bind(master** | **close** | **spread)** de la directive parallel
- Placer les threads 0,1,2,.. sur des coeurs voisins (les threads sont fixés sur un coeur)

  OMP_PLACES=cores

  OMP_PROC_BIND=close
- Placer les threads O,1,2,.. sur une même socket (les threads peuvent bouger)

  OMP_PLACES=sockets

  OMP_PROC_BIND=close
- Créer 2 équipes de threads chacune sur 1 processeur :

  OMP_PLACES=sockets

  OMP_PROC_BIND=spread,master

  #pragma OMP PARALLEL num_threads(2) proc_bind(spread)

  #pragma OMP PARALLEL num_threads(6) proc_bind(master)

Conclusion : optimiser l'exécution

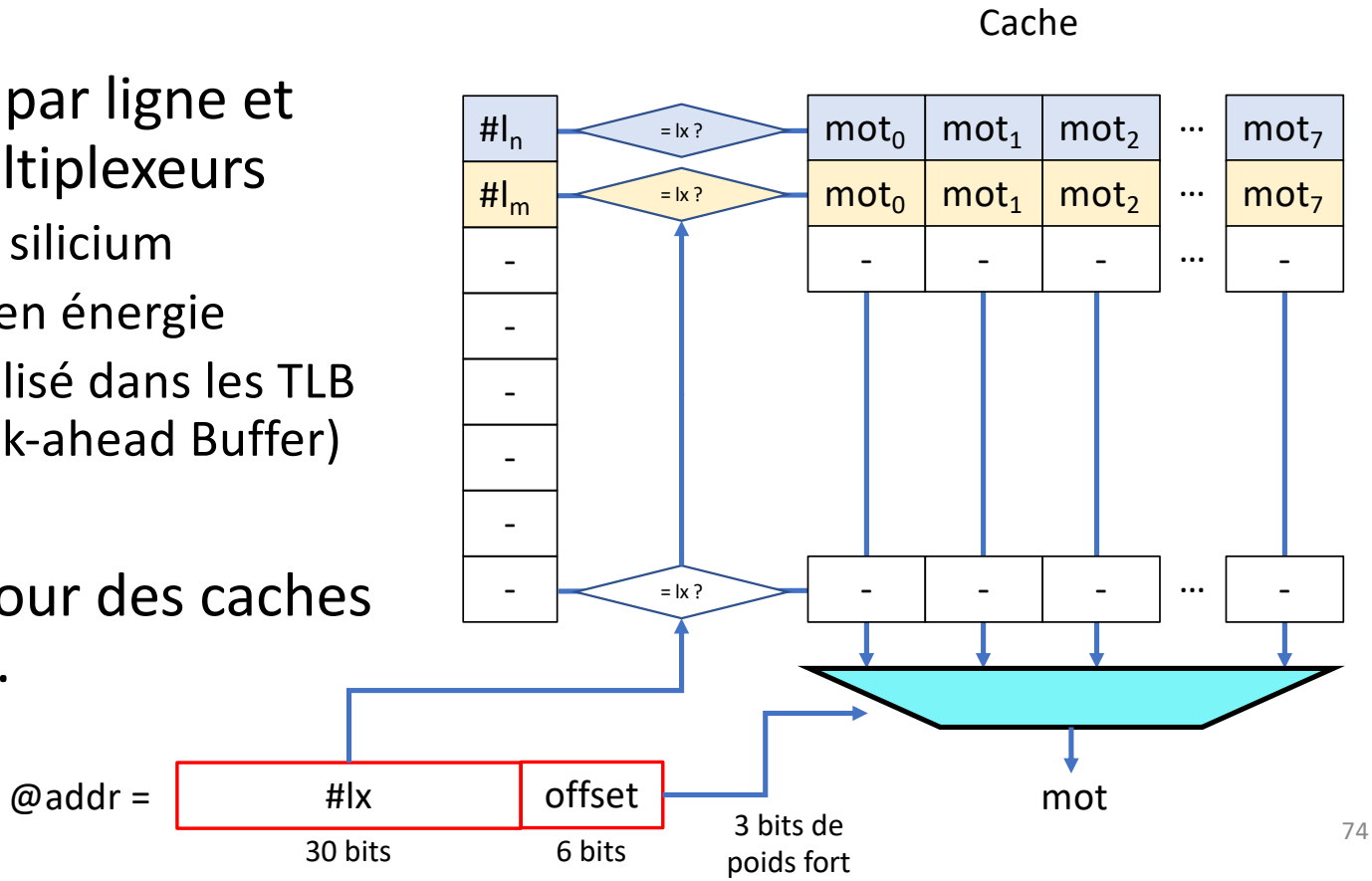
- Optimiser l'accès aux données
  - Optimiser l'utilisation du cache
  - Regrouper les données utilisées ensemble
    - Restructurer les données
  - Regrouper les traitements sur les mêmes données
    - Restructurer les boucles (tuiles,…)
    - Utiliser des algorithmes récursifs
      - cf. stratégie en profondeur d'abord d'OpenMP pour les tâches
  - Paralléliser l'accès aux données sur processeurs NUMA
- Optimiser l'utilisation du pipeline
  - Présenter du parallélisme d'instruction au sein des boucles
  - Remplacer les tests par du calcul

Tenir compte de l'architecture, du compilateur… et de la météo

Rab en vrac

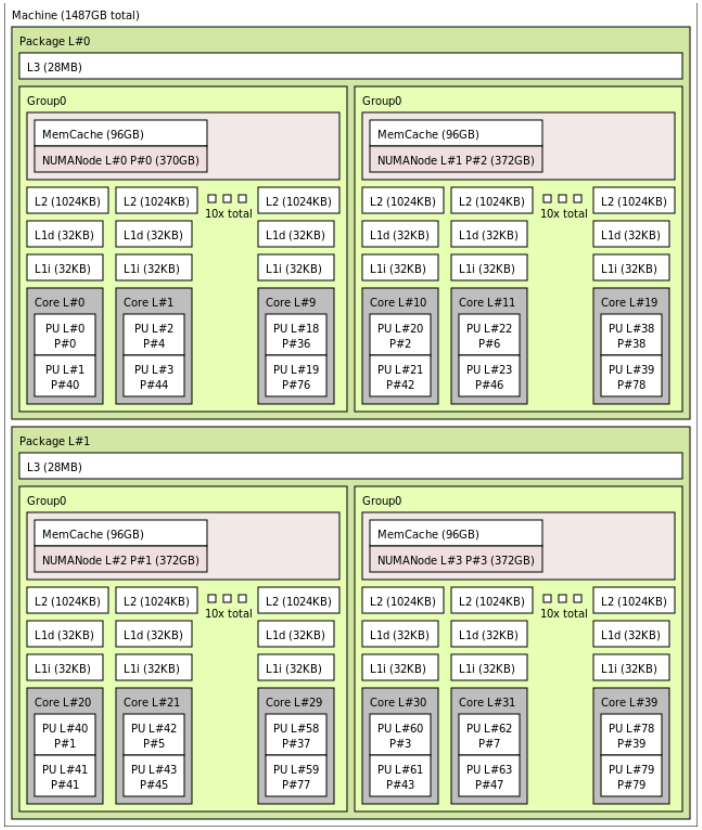
Fully Associative Cache

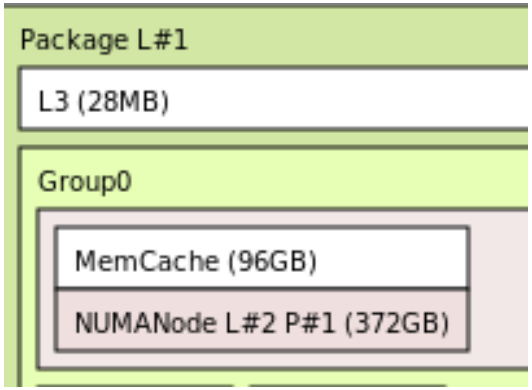
- Un comparateur par ligne et beaucoup de multiplexeurs
  - Très coûteux en silicium
  - Très gourmand en énergie
  - Usuellement utilisé dans les TLB (Translation Look-ahead Buffer)
- Trop complexe pour des caches de grande taille…

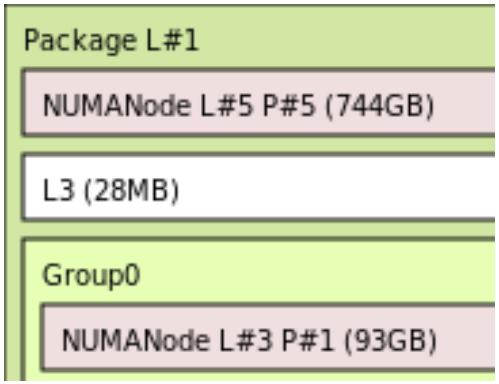


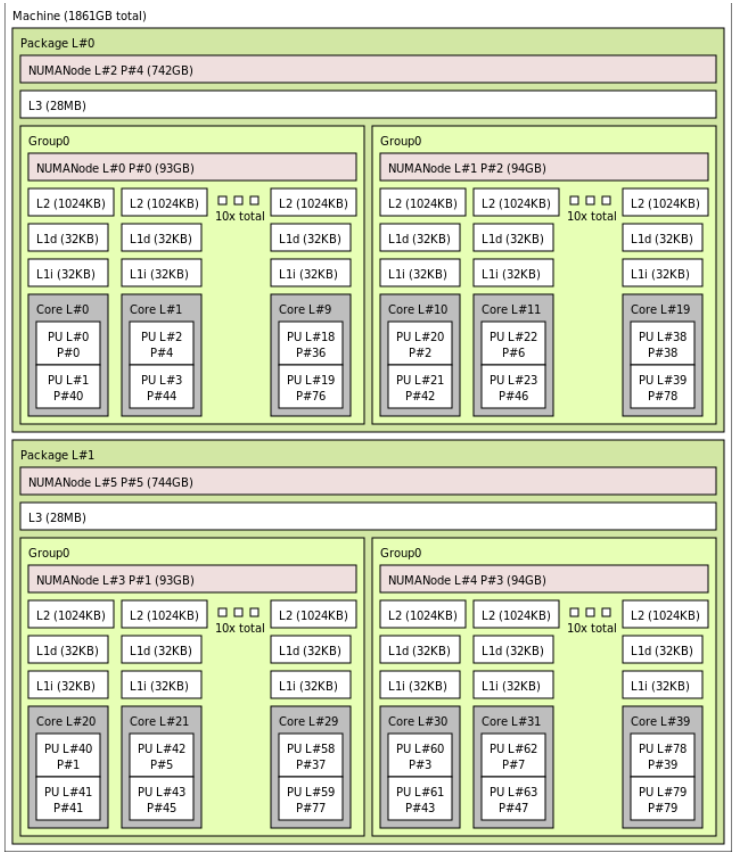
Processeurs configurables

2x Xeon *CascadeLake* 6230 with DDR as a cache in front of NVDIMMs



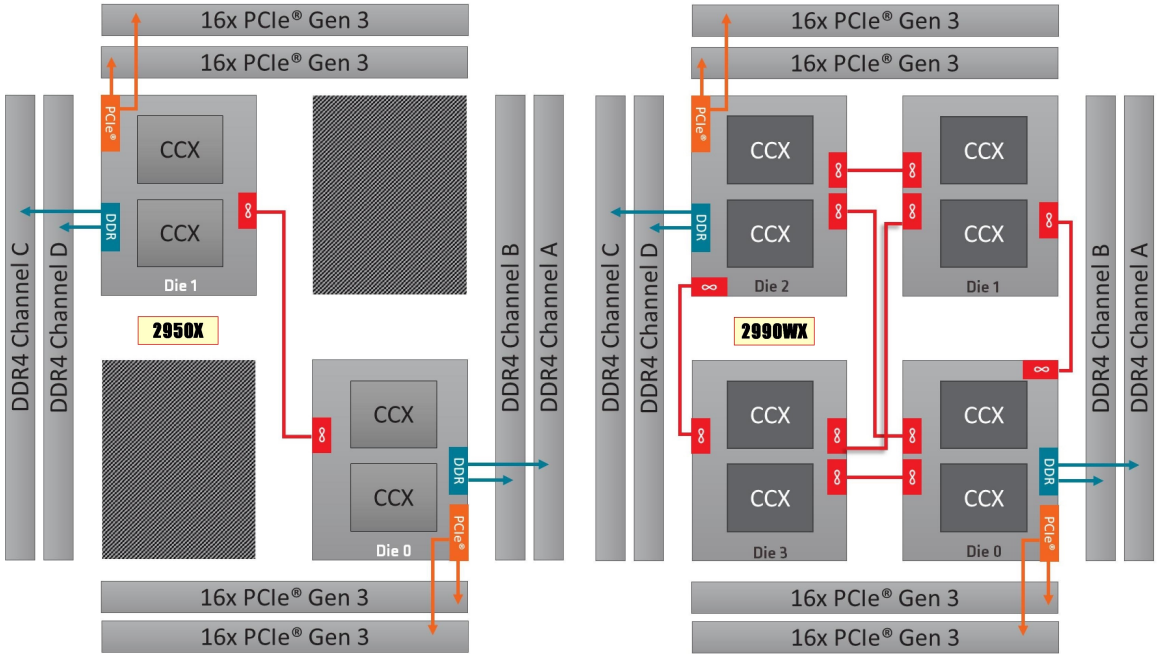






2x Xeon *CascadeLake* 6230 with NVDIMMs as separate NUMA nodes

AMD Threadripper – 2018
16 / 32 cœurs



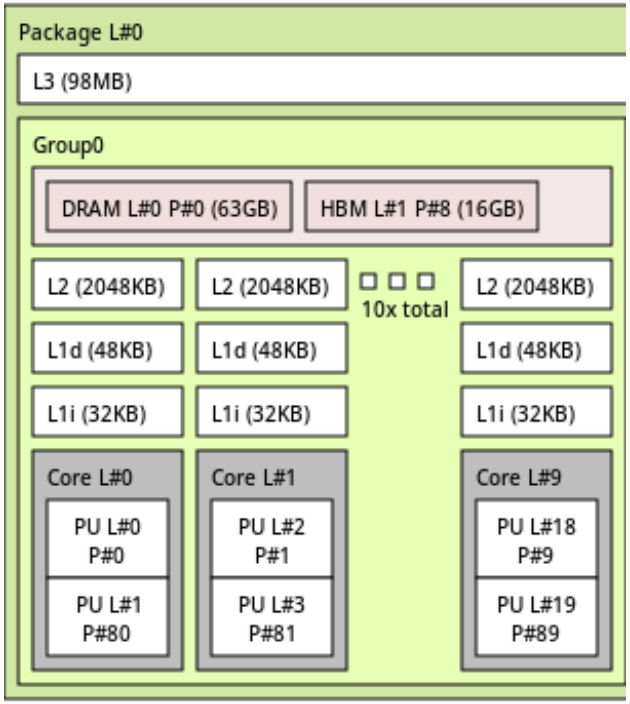
Placement des threads

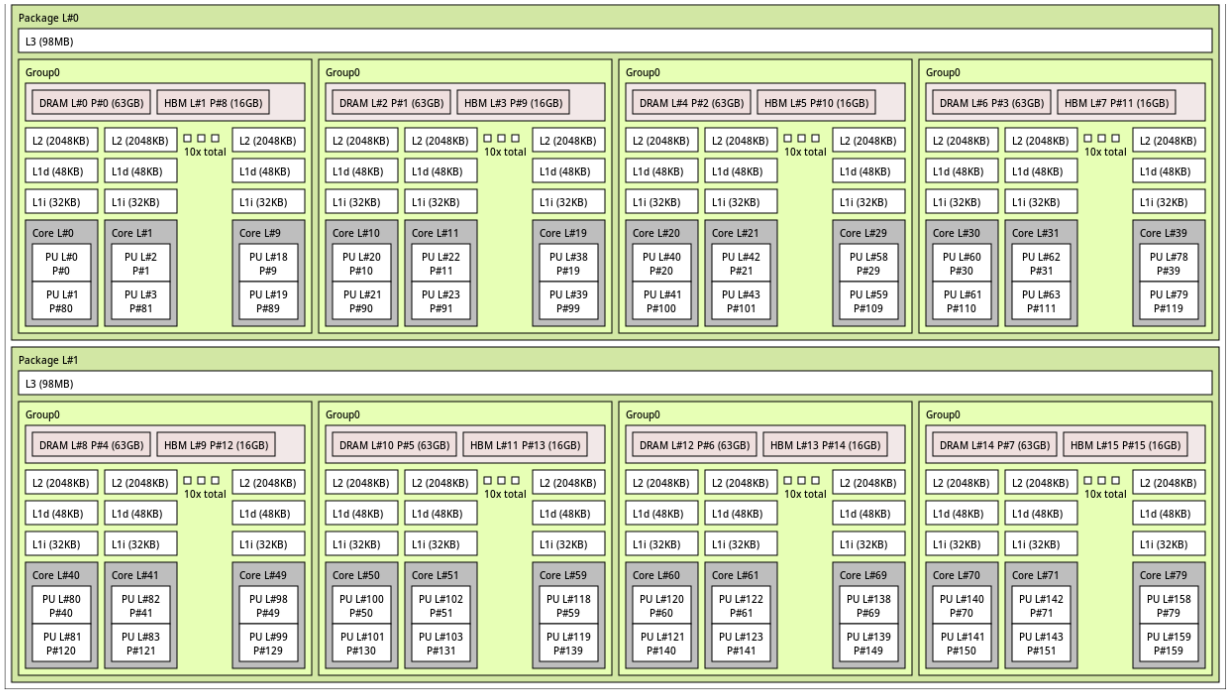
GNU / LINUX

Définition d'un masque précisant les numéros des cœurs où le thread pourra être exécuté

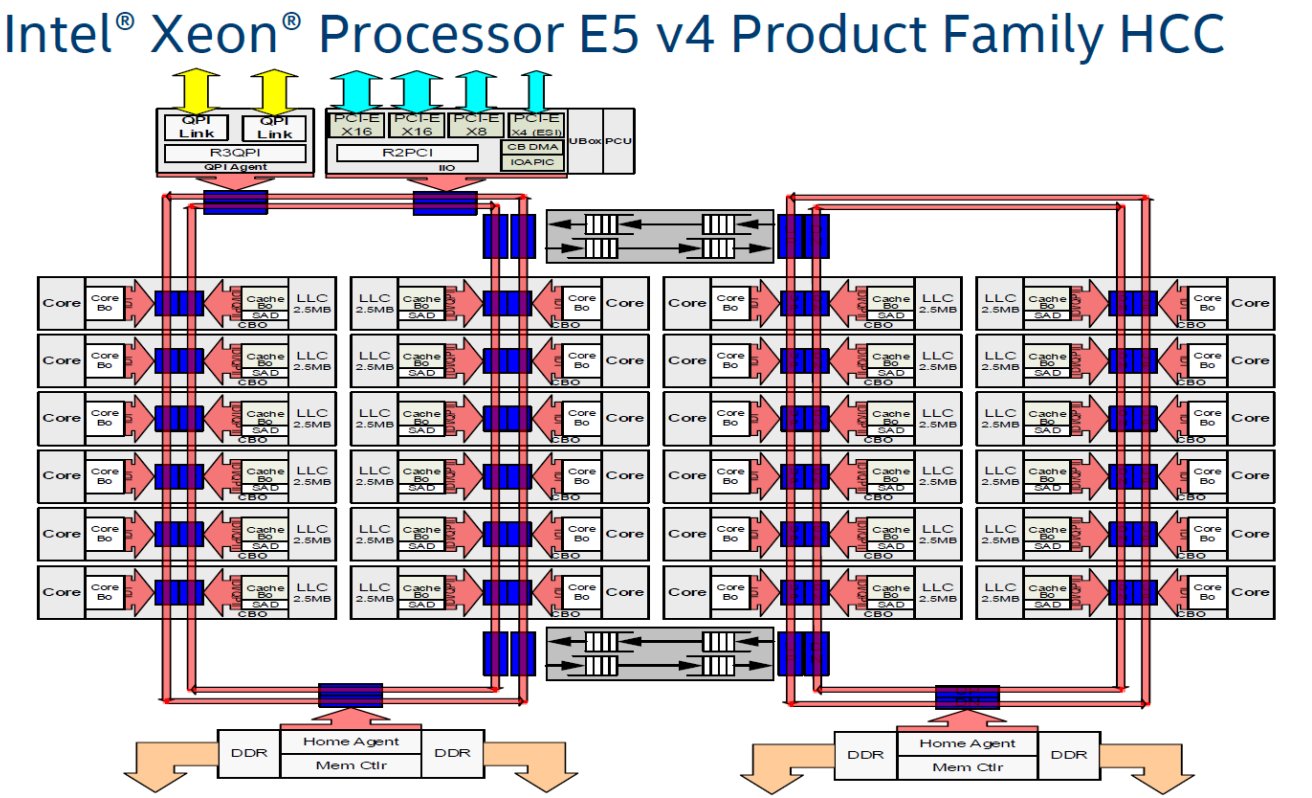
```
 int p[P];
pthread_t t[P];
pthread_attr_t attr[P];

for(i = 0; i<P; i++)
 {
   cpu_set_t cpuset;
   CPU_ZERO(&cpuset);
   CPU_SET(i%NB_COEURS,&cpuset);
   pthread_attr_init(&attr[i]);
   pthread_attr_setaffinity_np(&attr[i], sizeof cpuset, &cpuset);
   p[i]=i;
   pthread_create(&t[i],&attr[i],thread_function,&p[i]);
 }
```

2x Xeon *SapphireRapids* Max 9460 (from 2023, with hwloc v2.10).
Processors are configured in *SubNUMA-Cluster* mode, hence showing 4 DRAM NUMA nodes and 4 HBMs in each package.





Intel core I7 - 2016



Banc mémoire Banc mémoire Banc mémoire Banc mémoire