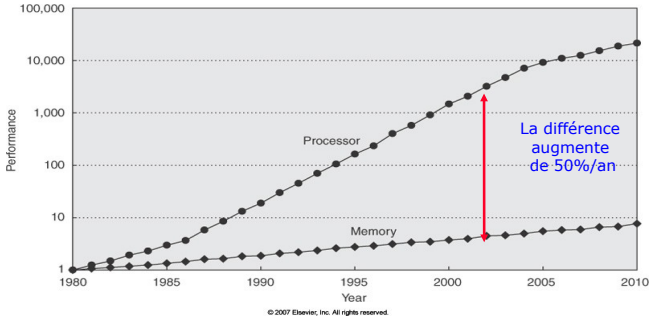
Les mémoires caches et optimisation pour la hiérarchie mémoire

K. Heydemann

Problème de la latence mémoire



~+60%/an jusqu'en 2004

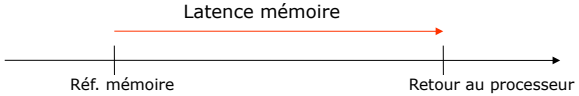
~+7%/an

Problématique :
temps de cycle processeur << temps d'accès mémoire.
Requête mémoire du processeur = processeur inactif plusieurs cycles.

1986 : temps de cycle du processeur ~ 120 ns
temps d'accès à la mémoire ~ 140 ns } 1

1996 : temps de cycle du processeur ~ 4 ns
temps d'accès à la mémoire ~ 60 ns } 20

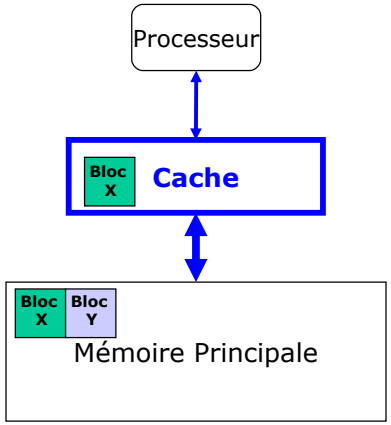
2002 : temps de cycle du processeur ~ 0.6ns
temps d'accès à la mémoire ~ 50 ns } 100



Masquer la latence mémoire : Mémoire Cache

- Mémoire cache : mémoire rapide (SRAM), entre 1 et 3 cycles (accès éventuellement pipeliné).
- Mais petite, car coûteuse (SRAM : 6 transistors par cellule, DRAM : 1 transistor par cellule).

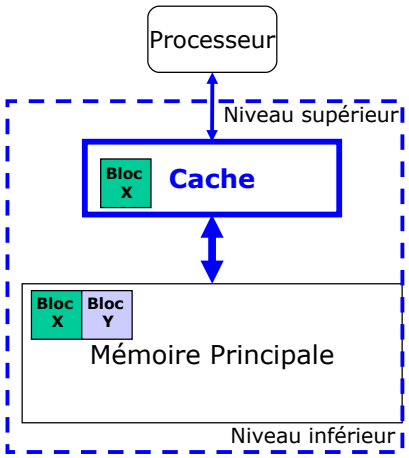
Technologie mémoire	Temps accès typique	$ par GB (2004)
SRAM	0.5 – 5 ns	$4K-$10K
DRAM	50 – 70 ns	$100-$200
Disque magnétique	5e6 – 20e6 ns	$0.50-$2



Fonctionnement des mémoires caches

Comment ça marche ?

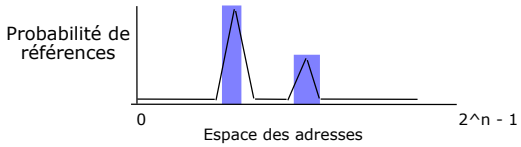
- Le processeur envoie ses requêtes mémoire au cache :
  - Si donnée dans le cache : **succès (*hit*)**
  - Si donnée hors du cache : **échec (*miss*)**
- Temps de succès << pénalité d'échecs.
- Temps de succès = temps d'accès + temps pour déterminer si succès ou échec.
- Temps d'échec = temps de succès + temps d'accès à la mémoire principale



Hiérarchie mémoire

Pourquoi ça marche ? Localité

- Principe de localité : à un instant donné, un programme accède à une portion relativement petite de l'espace d'adressage.

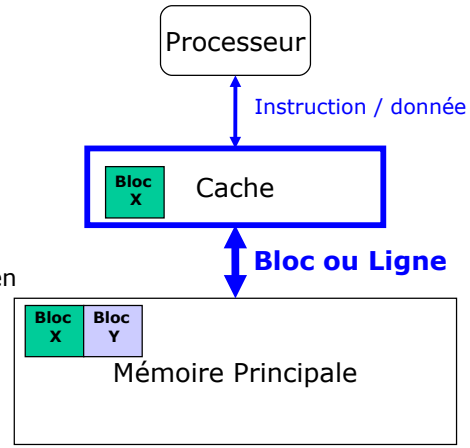


- **Localité temporelle** : adresse A référencée à T, alors forte probabilité de référencer A à T+t, t petit. Réutilisation des données.
- **Localité spatiale** : adresse A référencée, alors forte probabilité de référencer A+a, a petit. Utilisation de données voisines/contigües.

Exploitation de la localité

- Principe des caches : exploitation de la localité spatiale et de la localité temporelle.
- La localité temporelle est simplement exploitée en conservant une donnée dans le cache.
- La localité spatiale est exploitée en chargeant les données par **blocs** et non individuellement.

  Bloc : ensemble de données rangées consécutivement en mémoire



Localités des données et des instructions

Les instructions, comme les données, possèdent de fortes propriétés de localité.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    y[i] = y[i] + a[i][j] * x[j]
  }
}
```

- y[i]: propriétés de localités temporelle et spatiale.
- a[i][j]: propriété de localité spatiale.
- x[j]: propriété de localité temporelle et spatiale.

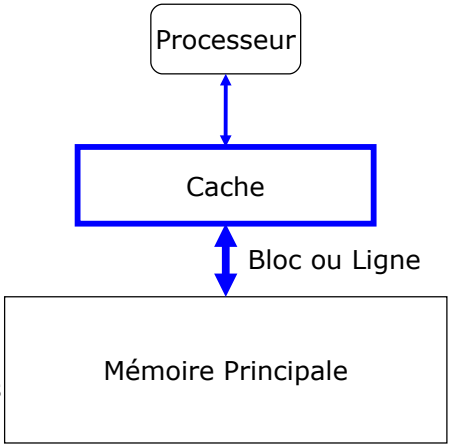
```
            ...
05 LOOP:   LD  R1, 3(R10)
06         ADDI R1, R1, 5
07         ST  R1, 30(R10)
08         ADDI R10, R10, 41
09         ADD R3, R0, R2
0A         BNE R10, R20, LOOP
            ...
```

Boucle : réutilisation des instructions : localité temporelle

Instructions consécutives en mémoire : localité spatiale

Caractéristiques d'un cache

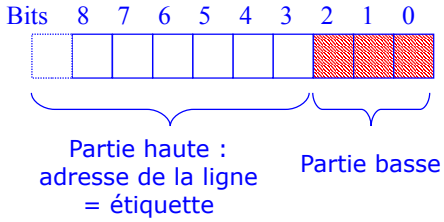
- Taille de la ligne = taille du bloc.
- Taille du cache.
- Organisation du cache (associativité, politique de placement).
- Politique d'écriture (politique de gestion de la cohérence mémoire/cache et de mise en cache ou non des données écrites).



Bloc ou ligne de cache

- Taille des transferts entre la mémoire et le cache.
- Mémoire est donc découpée en blocs.
- Adresse vue par le processeur = etiquette/adresse du bloc en mémoire + déplacement dans ce bloc :
  - la partie haute de l'adresse des données d'une même ligne est identique,
  - seule la partie basse de l'adresse varie : designe les données dans le bloc

Exemple :
•Adresse sur 16 bits
•Bloc de 8 octets



0...010100000
0...010100111
0...010101000

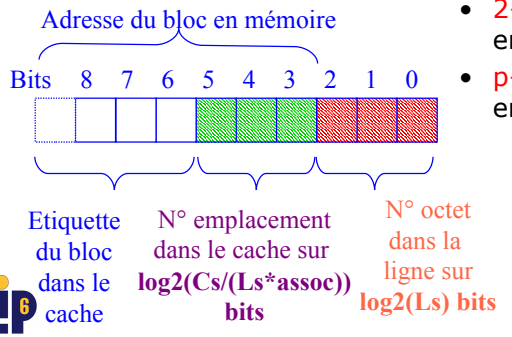
Même ligne de cache
Lignes distinctes, adresses consécutives

Organisation du cache

- Si un cache a :
  - une taille CS
  - une taille de ligne LS
  - => possède CS/LS lignes.
  - Stockage de n = CS/LS blocs mémoire différents à un instant donné.
- Placement des blocs dans ces n lignes ?
  - Le placement des données dans le cache est géré par le matériel
  - le programmeur n'a pas à se soucier du placement des données (contrairement à mémoire locale),
  - le fonctionnement du cache est transparent pour le programmeur

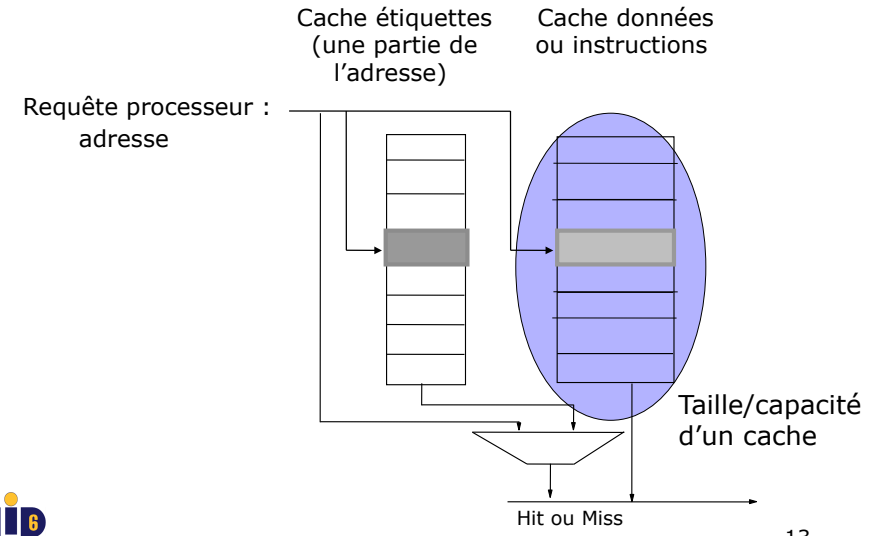
Placement des données dans le cache et associativé

- L'emplacement d'une donnée est déterminé selon une fonction simple de l'adresse.
  - N° emplacement dans le cache
  - N° d'octet dans la ligne

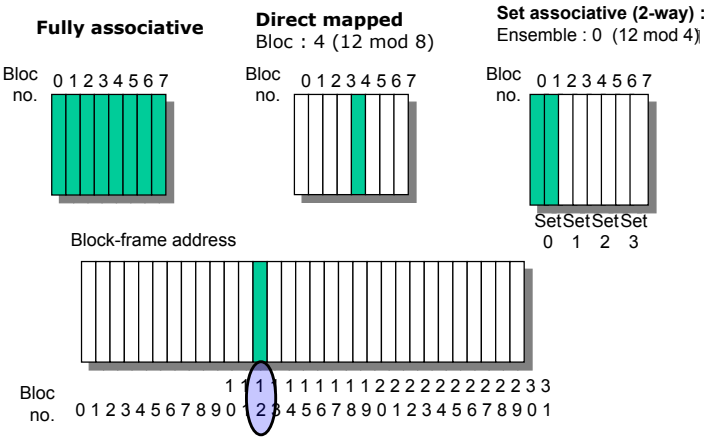


- L'associativité = nombre de ligne par emplacements
- Correspondance directe ou assoc = 1. 1 seule ligne pour une donnée. N lignes = N emplacements différents
- 2-associatifs : N/2 emplacements de 2 lignes.
- p-associatifs : N/p emplacements de p lignes

Structure générale d'un cache

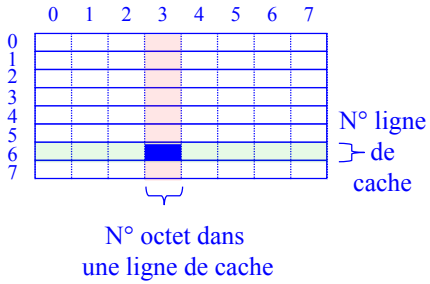


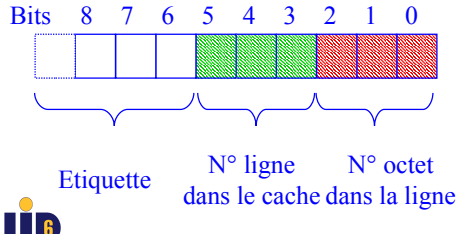
Différentes organisations de cache



Placement des données – Cache à correspondance directe

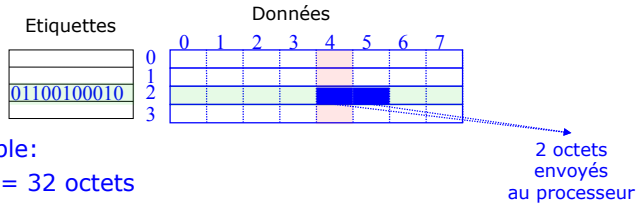
- 1 seule ligne/emplacement donc 1 seule possibilité pour une donnée
- L'adresse de la donnée donne donc :
  - N° ligne dans le cache
  - N° d'octet dans la ligne





- Cache de C_S octets
- Ligne de L_S octets
- N° octet: $\log_2(L_S)$ bits de poids faible de l'adresse.
- N° ligne: $\log_2(C_S/L_S)$ bits de poids faible suivants de l'adresse.

Lecture d'une donnée – Cache à correspondance directe



- Exemple:
  - C_S = 32 octets
  - L_S = 8 octets
- Adresse demandée (16 bits):
  - 0110010001010100
  - N° ligne: 10
  - N° octet dans la ligne: 100
- Une requête peut avoir une taille variable:
  - octet, demi-mot, mot...
  - requête = adresse + nombre d'octets
  - adresse = adresse du premier octet
  - exemple: 2 octets (mot de 16 bits)

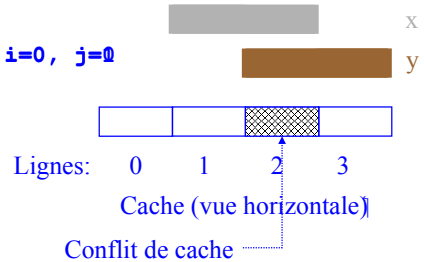
Intérêt de l'associativité

- Taille mémoire physique >> taille cache.
- La fonction de placement peut engendrer des conflits entre les données.
- CS = 32 o, Ls = 8 o, x et y flottants double précision (8 o).

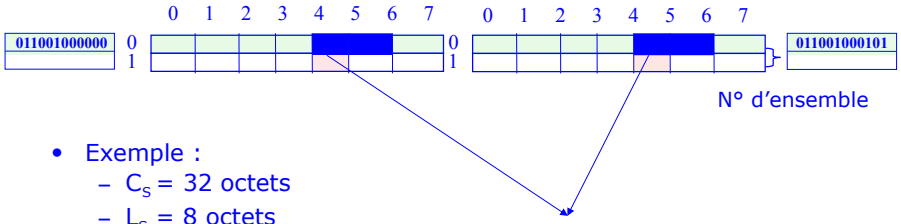
On peut réduire les conflits en augmentant l'**associativité** des caches.

```
for (i=0; i<N; i++) {
  for (j=0; j<2; j++) {
    x[j] = y[j]
  }
}
```

```
@x = 0110010001001000
@y = 0000100000010000
```



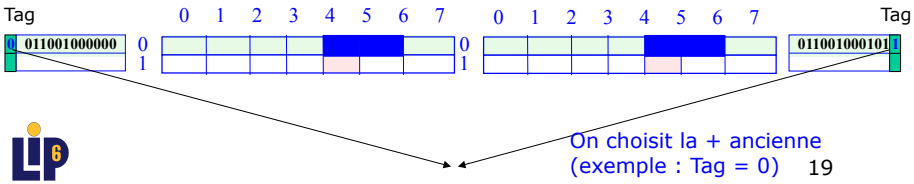
Lecture d'une donnée – Cache associatif



- Exemple :
  - C_s = 32 octets
  - L_s = 8 octets
  - A = 2
  - Requête = 2 octets
- Adresse demandée (16 bits) :
  - 0110010001010100
  - N° d'ensemble : **0**
  - N° octet dans la ligne: **100**

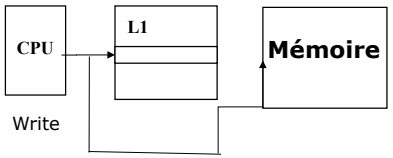
Placement d'une donnée - Cache associatif

- Une donnée peut être stockée dans *n* emplacements différents.
- Il faut choisir la ligne dans laquelle on va charger la nouvelle donnée.
- Le choix est effectué entre les lignes du cache qui peuvent accueillir la donnée (l'ensemble):
  - LRU (*Least Recently Used*) : on choisit le bloc le plus anciennement accédé.
  - FIFO (*First In First Out*).
  - *Random.*
  - Pseudo-LRU : la ligne la plus récemment accédée n'est pas remplacée ; choix aléatoire entre les autres lignes.



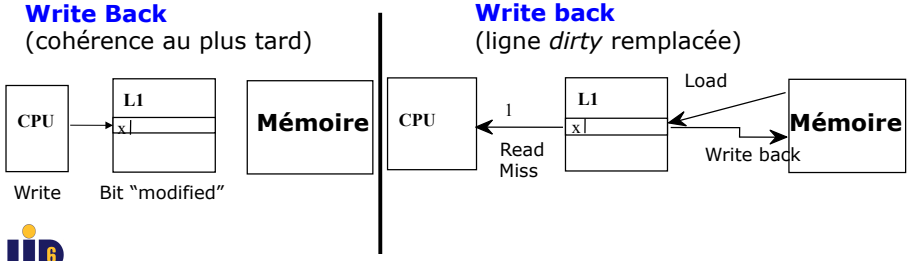
Politiques d'écriture en cas de succès

Write Through
(cohérence au plus tôt)

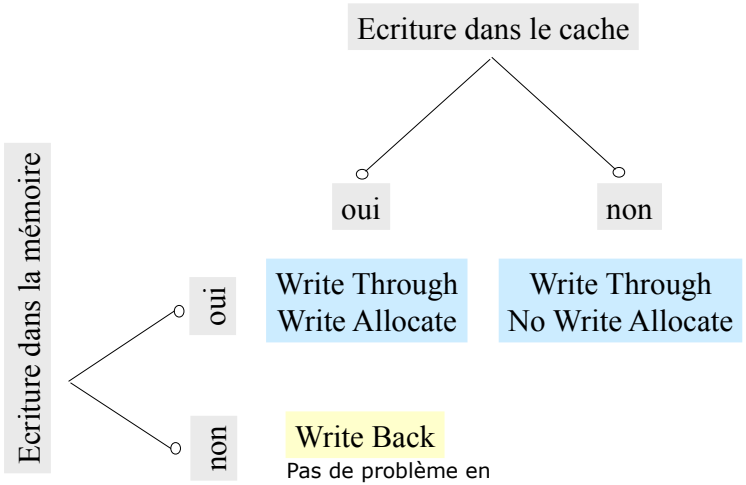


Write Back
(cohérence au plus tard)

Write back
(ligne *dirty* remplacée)



Politique d'écriture en cas d'échec



Pas de problème en monoprocesseur – Problème de cohérence en multiprocesseurs.

Impact sur la performance

- Performance :
  - CPUTime = (Temps-ALU + AMAT) * Temps de cycle
- Temps d'accès moyen à la mémoire :
  - AMAT = Temps de succès + Taux d'échecs * Pénalité d'échecs

Impact sur la performance (suite)

- Hypothèses :
  - Variation du nombre d'échecs pour les instructions et les données : 100%, 50%, 10%, 1%, 0%.
  - Pénalité d'échec 100 cycles
  - Accès en 1 cycle si succès

1. AMAT = 1+1x100 = 101 2. AMAT = 1+0.5x100 = 51 3. AMAT = 1+0.1x100 = 11 4. AMAT = 1+0.01x100 = 2 5. AMAT = 1+0x50 = 1	S'il faut 100 cycles pour un Programme avec 20 inst. mem 1. NbCycle = 100 + 20*(101-1) = 2100 2. NbCycle = 100 + 20*(51-1) = 1100 3. NbCycle = 100 + 20*(11-1) = 300 4. NbCycle = 100 + 20*(2-1) = 120 5. NbCycle = 100 + 20*(1-1) = 100

Il est important de minimiser le taux d'échec ou la pénalité d'échec !

Réduire la pénalité d'échecs

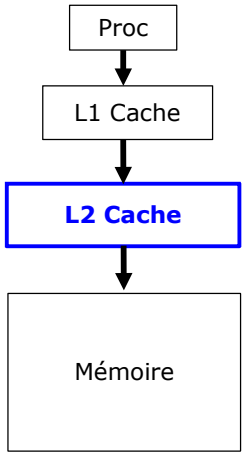
Cache de second niveau

- Comment masquer plus la latence mémoire ? Ajouter des mémoires cache entre le cache et la mémoire : plusieurs niveaux de cache L1, L2, L3...
- Second niveau de cache plus grand - meilleur taux de succès. Temps de succès << pénalité d'échec.

$AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} * Miss\ Penalty_{L1}$

$Miss\ Penalty_{L1} = Hit\ Time_{L2} + Miss\ Rate_{L2} * Miss\ Penalty_{L2}$

$AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} * (Hit\ Time_{L2} + Miss\ Rate_{L2} * Miss\ Penalty_{L2})$



Exemple de calcul de taux de miss

```
int A[N];
for(i = 0; i < N; i++)
   S+= A[i];
```

Si Cs = 1Ko Ls = 32o
Nb accès = N
Nb miss = ?
Taux d'échec = ?

```
int A[N];
for(j = 0 ; j < N ; j++)
  for(i = 0; i < N; i++)
       S+= A[i];
```

Si Cs = 1Ko Ls = 32o
Nb accès =
Si N vaut 64, 128, 256
Nb miss = ?
Taux d'échec = ?
Si N est > 256 ?
Nb miss = ?

Optimisations logicielles des mémoires caches

Les types d'échecs : les 3Cs
(un peu de vocabulaire)

- **Compulsory misses** ou miss de démarrage. Le premier accès à un bloc absent du cache. Echec même en cas de cache infini.
- **Capacity misses** ou miss de capacité. Echecs sur une même donnée dû à un ensemble de données manipulées trop grand. Echec même en cas de cache complétement associatif.
- **Conflict ou interference misses** ou miss de conflit. Echec du à l'utilisation d'un même ensemble (bloc). Degré d'associativité insuffisant.
- **Un 4ème C : Coherence**. Echec causé par la gestion de la cohérence de cache.

Des optimisations à la compilation

Objectif : réduire le taux d'échecs.
- augmenter la localité spatiale et temporelle des codes.
- éliminer les conflits entre les données.
- vérifier que le code est correct...

- Regroupement de tableaux (*merging arrays*),
- Permutation de boucles (*loop interchange*),
- Fusion de boucles (*loop fusion*),
- Renversement de boucles (*loop reversal*),
- Fission de boucles (*loop fission*),
- *Blocking*,
- *Padding*,
- Préchargement (charger plus tôt),
- …

La plupart des optimisations pour les caches réalisées pour des boucles et des tableaux.

Regroupement de tableaux

/* **Avant** : 2 tableaux séquentiels */

```
int val[SIZE];
int key[SIZE];
```

/* **Après** : 1 tableau de structures */

```
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

```
int a[N], b[N], c[N];
for (i = 0; i<N; i++)
  {a[i] = b[i] + c[i];}
```

```
struct ALL {int a, b, c};
struct ALL M[N];
for (i = 0; i<N; i++)
  {M[i].a = M[i].b + M[i].c;}
```

- Réduction des conflits entre val & key ; augmentation de la localité spatiale.

Permutation de boucles

- Changer l'ordre des boucles afin d'accéder aux données dans l'ordre dans lequel elles sont stockées en mémoire.

/* **Avant** */

```
for (k = 0; k < 100; k = k+1)
   for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];
```

/* **Après** */

```
for (k = 0; k < 100; k = k+1)
   for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

- Augmentation de la localité spatiale. N grand.

Fusion de boucles

- Combiner des boucles indépendantes (avec même compteur et des variables identiques).

/* **Avant** */

```
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
    a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
    d[i][j] = a[i][j] + c[i][j];
```

/* **Après** */

```
for (i = 0; i < N; i = i+1)
   for (j = 0; j < N; j = j+1)
   {a[i][j] = 1/b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j];}
```

Un exemple plus simple :

```
for (i = 0; i<N; i++)
   a[i] = b[i] + 1;
for (i = 0; i<N; i++)
   c[i] = 3*b[i];
```

```
for (i = 0; i<N; i++)
  { a[i] = b[i] + 1;
    c[i] = 3*b[i];}
```

- Augmentation de la localité temporelle. N grand – pas de localité temporelle dans la boucle.

Renversement de boucle

```
for (i = 0; i<N; i++) {
   a[i] = b[i] + 1;
   c[i] = 3*a[i];
}
for (i = 0; i<N; i++)
   d[i] = c[i+1] +1;
```

```
for (i = N; i>= 0; i--) {
   a[i] = b[i] + 1;
   c[i] = 3*a[i];
}
for (i = N; i>= 0; i--)
   d[i] = c[i+1] +1;
```

```
for (i = N; i>= 0; i--) {
   a[i] = b[i] + 1;
   c[i] = 3*a[i];
   d[i] = c[i+1] +1;
}
```

Rend les boucles fusionnables ici.

Fission de boucle

```
for (i = N; i>= 0; i--){
   a[i] = b[i] + 1;
   c[i] = 3*a[i];
   f[i] = g[i] +h[i];
  }
```

```
for (i = 0; i<N; i++) {
   a[i] = b[i] + 1;
   c[i] = 3*a[i];
}
for (i = 0; i<N; i++)
    f[i] = g[i] +h[i];
```

- Augmentation de la localité temporelle – réduire les conflits.

Array padding

/* Avant */

```
int a[N], b[N];
   for (i = 0; i < N; i++)
     sum+ = a[i] * b[i];
```

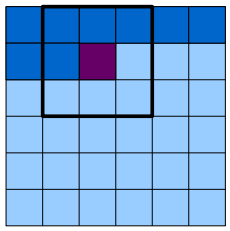
/* Après */

```
int a[N], pad [x], b[N];
   for (i = 0; i < N; i++)
     sum+ = a[i] * b[i];
```

- N grand – N multiple de la taille du cache (ex : direct-mapped).
- Eviter les miss de conflit.
- Ex : x = 8 pour une taille de ligne de 32 octets.

Introduction au blocking

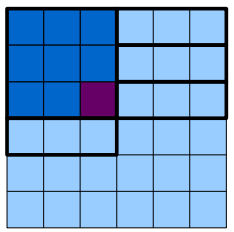
- Stencil : calcul de la (nouvelle) valeur d'un point (i,j) en fonction du voisinage
- Tab2[i][j] = Tab[i-1][j-1] + Tab[i-1][j] +Tab[i-1][j+1] + Tab [i][j-1] + Tab[i][j] +Tab[i][j+1] + Tab[i+1][j-1] + Tab[i+1][j] + Tab[i+1][j+1]

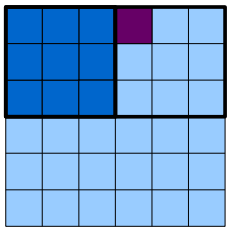


Si la matrice est volumineuse (ex : taille de ligne = taille de cache) Les données ne sont plus dans le cache quand on change de ligne => travailler par bloc permet d'exploiter les données charger dans le cache

Blocking

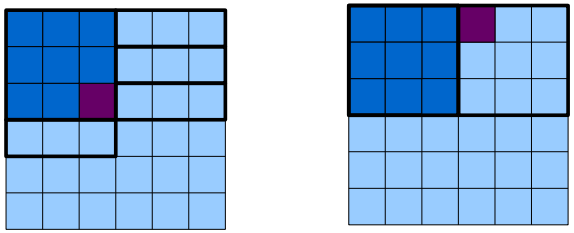
- Tab2[i][j] = Tab[i-1][j-1] + Tab[i-1][j] +Tab[i-1][j+1] + Tab[i][j-1] + Tab[i][j] +Tab[i][j+1] + Tab[i+1][j-1] + Tab[i+1][j] + Tab[i+1][j+1]
- Calcul sur taille de bloc tenant largement dans le cache (blocs de cache avec les données en dessous et sur le coté chargés...)





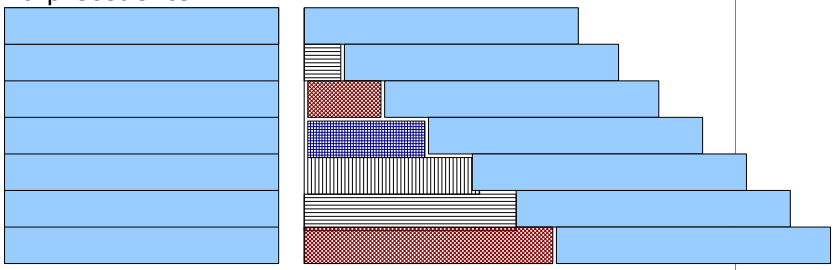
Blocking & padding...

- Tab'[i][j] = Tab[i-1][j-1] + Tab[i-1][j] +Tab[i-1][j+1] + Tab'[i][j-1] + Tab[i][j] +Tab[i][j+1] + Tab[i+1][j-1] + Tab[i+1][j] + Tab[i+1][j+1]
- Problème si une ligne a une taille multiple de la taille du cache et si cache à correspondance directe : les données d'un bloc sont en conflit dans le cache...
- Padding possible... mais plus complexe !



Blocking & padding...

- Problème si une ligne a une taille multiple de la taille du cache et si cache à correspondance directe : les données d'un bloc sont en conflit dans le cache...
- Padding possible... mais plus complexe !
- Décaler chaque ligne (de la taille d'un bloc) par rapport à la précédente...



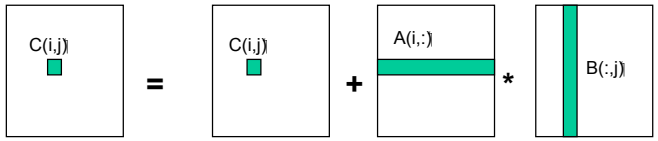
- Plus de mémoire, mais pas/moins de conflit dans le cache :-)

Multiplication de matrices

```
/* Calcul de C = C + A*B */
for i = 1 to n
  for j = 1 to n
   for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Réutilisation sur j

Réutilisation sur i

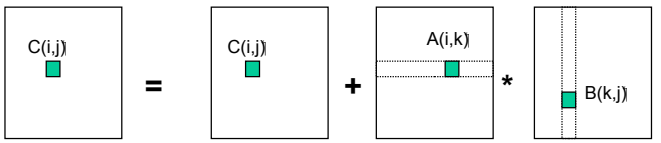


- Si le cache ne peut pas contenir B entièrement, il y aura un miss à chaque accès.
- Si le cache ne peut pas contenir une ligne de A entièrement, il y aura un miss à chaque ligne.

Blocking

- Adapter la granularité à la hiérarchie mémoire – décomposer le problème en sous-problème tenant dans le cache.
- Calculer par blocs :on considère que les matrices A, B et C sont des matrices N x N composées de sous-blocs b x b où b = n / N

```
for i = 1 to N step b
   for j = 1 to N step b
       for k = 1 to N step b
        {block A(i,k)-A(i+b,k+b) into fast memory}
        {block B(k,j)- B(k+b,j+b) into fast memory}
        {block C(i,j)-C(i+b,j+b) into fast memory}
       {do a matrix multiply on blocks}
 {write block C(i,j) back to slow memory}
```

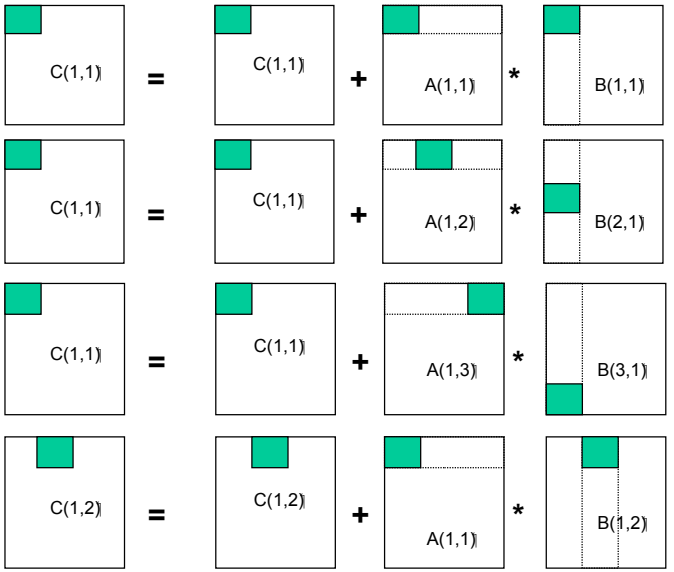


Blocking

- Adapter la granularité à la hiérarchie mémoire – décomposer le problème en sous-problème tenant dans le cache.
- Calculer par blocs :on considère que les matrices A, B et C sont des matrices N x N composées de sous-blocs b x b où b = n / N

```
for i = 1 to N step b
  for j = 1 to N step b
    for k = 1 to N step b
      for ii = i to i + b
        for jj = j to j + b
          for kk = k to k + b
            C(ii,jj) = C(ii,jj) + A(ii,kk)*B(kk,jj)
```


Multiplication matrices bloquées



Avec C(i,j), A(i,j), B(i,j) bloc de taille b*b

Préchargement logiciel

- Charger les données en avance, avant que le processeur en ait besoin
- Permet d'éviter les miss de démarrage
- Les jeux d'instructions comportent des instructions spéciales permettant de lancer le préchargement (chargement qui s'arrête au cache).
- Instructions générées par le compilateur à la suite d'analyse de dépendances et de flot.

/*Avant */

```
for i = 1,n do
   A(i) = A(i) + val * B(i)
```

/*Après*/

```
Prefetch A(1), B(1), and
for i = 1,n do
   A(i) = A(i) + val * B(i)
   Prefetch A(i+k), B(i+k)    /* inutile à toutes les itérations*/
                              /*toutes les LS/sizeof(A[i]) suffit*/
```


Effet des optimisations