Optimisation de code pour la hiérarchie mémoire

K. Heydemann

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.

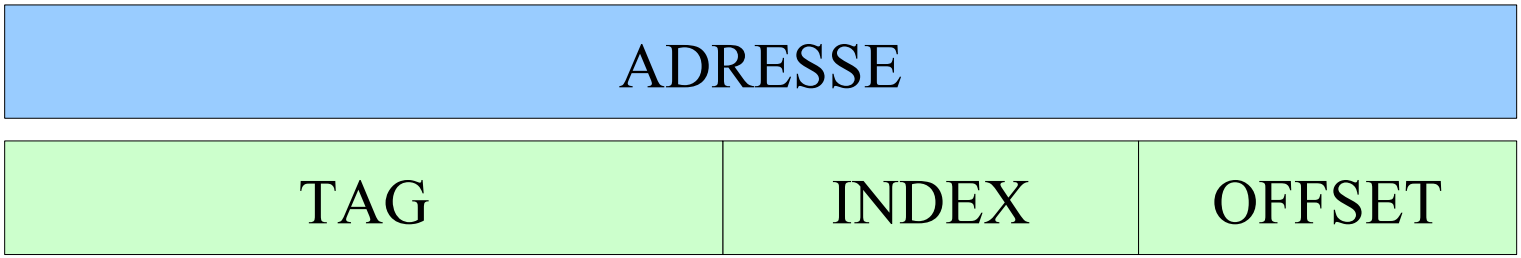
Rappels

- Localité temporelle :
  si `D` utilisée à l'instant `t` réutilisation dans un futur proche `t + delta(t)` de `D`

- Localité spatiale :
  si `D` d'adresse `A` utilisée à l'instant `t` réutilisation dans un futur proche `t + delta(t)` de `A + delta(A)`

Rappels

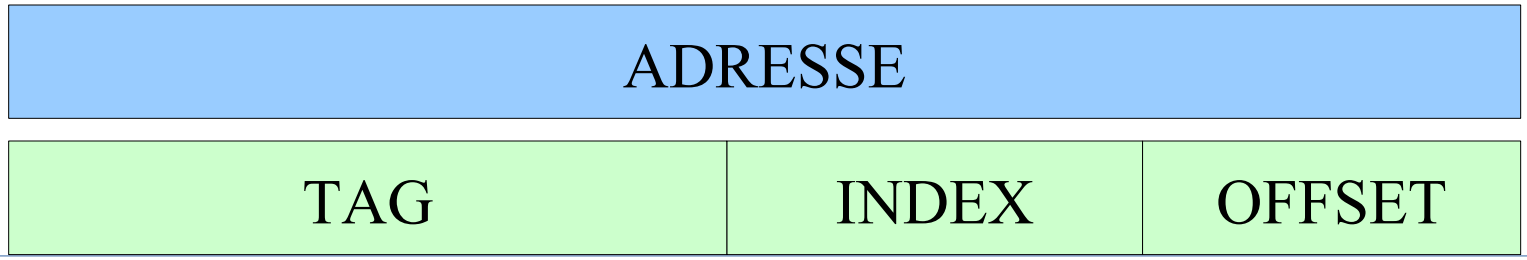
- Le placement d'une donnée D dans le cache dépend de son adresse A et des caractéristique du cache
- Cache = taille N octets, ligne L octets, associativité a



- Nb bits offset = log2(L)
- Nb bits d'index = log2(nb d'ensembles)
  = log2(N÷ (L x a))

Rappels

- Deux données X et Y vont dans le même ensemble du cache si elles ont le même index
- Dans le même emplacement si même index et même offset i.e que leurs adresses modulo 2^(N÷a) sont égales i.e adresse multiple de (N÷a)
- On dit qu'elles sont alignées, si ce sont des tableaux alors X[i] et Y[i] vont dans les mêmes emplacements pour tout i
- Conflits possibles si utilisées en même temps et cache direct-mapped



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*,
- …

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];
```

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

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

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

```
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 :

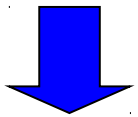
```
for (i = 0; i<N; i++)
   a[i] = b[i] + 1;
for (i = 0; i<N; i++)
   c[i] = 3*b[i];

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

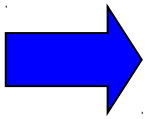
- Augmentation de la localité temporelle. N grand – pas de localité temporelle dans la boucle.

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;
```

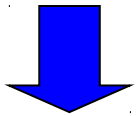


```
for (i = N; i>= 0; i--) {
   a[i] = b[i] + 1;
   c[i] = 3*a[i];
   d[i] = c[i+1] +1;
}
```

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

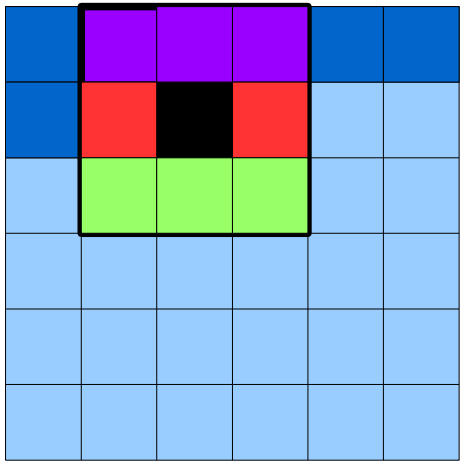
```
/* Avant */
int a[N], b[N];
   for (i = 0; i < N; i++)
     sum+ = a[i] * b[i];
```

```
/* 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



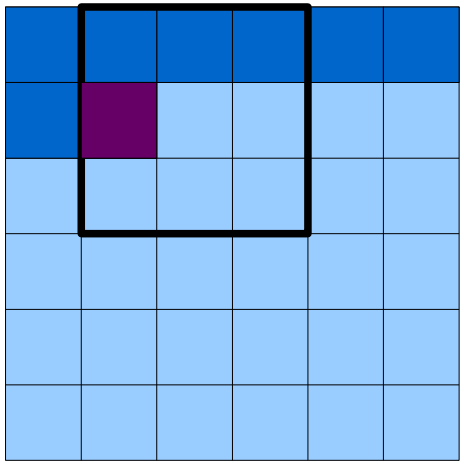
```
for (i = 0 ; i<N ; i++) {
   for (j = 0 ; j<N ; j++)
    tab2[i][j] = tab[i-1][j-1] + tab[i-1][j]
               + tab[i-1][j+1] + tab [i][j-1]
               + tab[i][j] +tab[i][j+1]
               + tab[i+1][j-1] + tab[i+1][j]
               + tab[i+1][j+1]

}
```

Introduction au blocking

- Stencil : calcul de la (nouvelle) valeur d'un point (i,j) en fonction du voisinage
- tab2[i][j] =

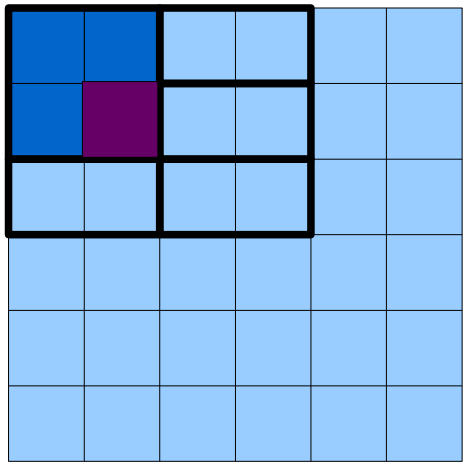
tab[i-1][j-1] + tab[i-1][j] +tab[i-1][j+1]
+ tab [i][j-1] + tab[i][j] +tab[i][j+1]
+ tab[i+1][j-1] + tab[i+1][j] + tab[i+1][j+1]

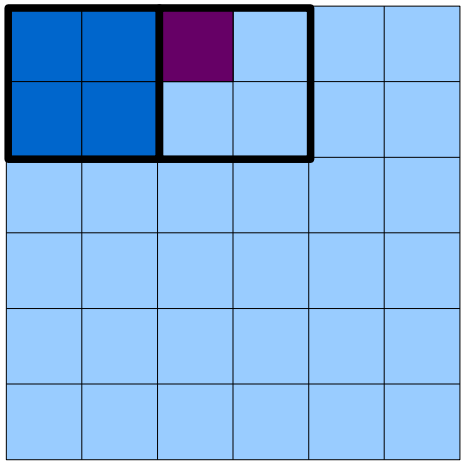


Si la matrice est volumineuse
(ex : taille de ligne = taille de cache)
Les données ne sont plus dans le cache quand on change de ligne
=> travailler par bloc permet d'exploiter les données charger dans le cache

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...)

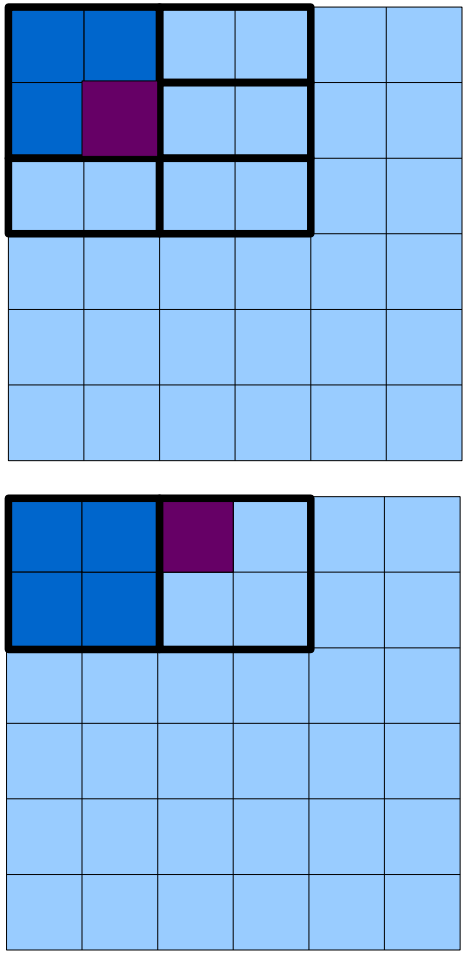




Blocking

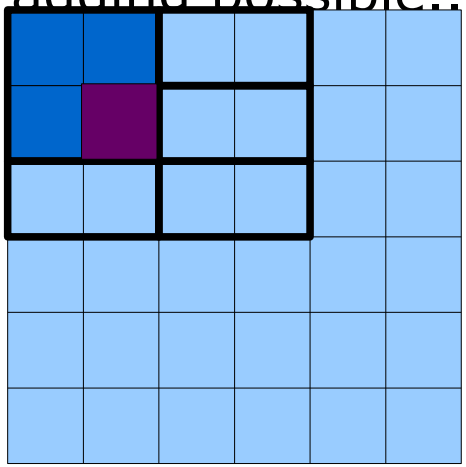
Calcul sur taille de bloc tenant largement dans le cache (blocs de cache avec les données en dessous et sur le coté chargés...)

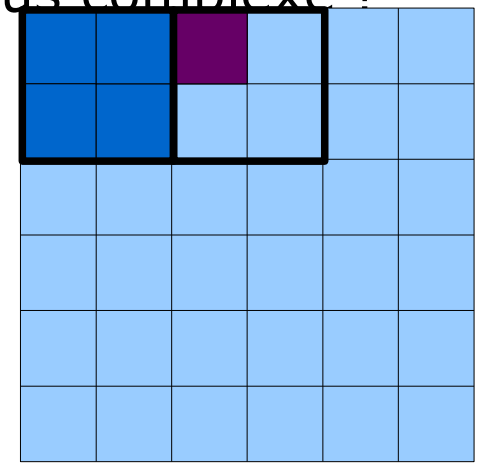
```
for (i = 0 ; i<N ; i+= B) {
  for (j = 0 ; j<N ; j+=B) {
    for (ii=i ; ii<i+B ; ii++) {
      for (jj = j ; jj < j+B ; jj++){
        tab2[ii][jj] = tab[ii-1][jj-1]
                     + tab[ii-1][jj] + tab[ii-1][jj+1]
                     + tab [ii][jj-1] + tab[ii][jj]
                     + tab[ii][jj+1] + tab[ii+1][jj-1]
                     + tab[ii+1][jj] + tab[ii+1][jj+1]
        }
      }
    }
}
```



Blocking & padding...

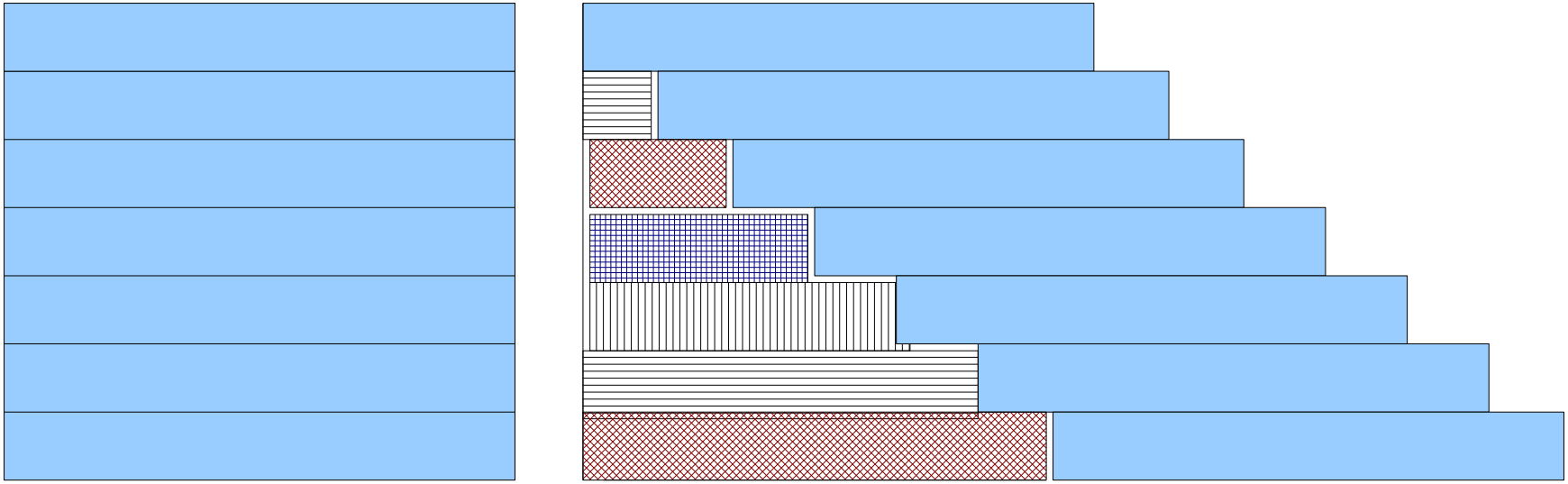
- Tab2[i][j] = Tab[i-1][j-1] + Tab[i-1][j] +Tab[i-1][j+1]
  + Tab[i][j-1] + Tab[i][j] +Tab[i][j+1]
  + Tab[i+1][j-1] + Tab[i+1][j] + Tab[i+1][j+1]
- Problème si une ligne a une taille multiple de la taille du cache et si cache à correspondance directe : les données d'un bloc sont en conflit dans le cache...
- 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 nécessaire, mais pas/moins de conflits dans le cache :-)