Mineure HPC-SBD

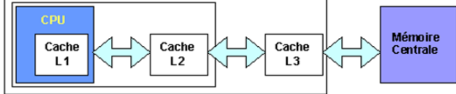
# Optimisations sérielles et vectorisation


**Stéphane Vialle**

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

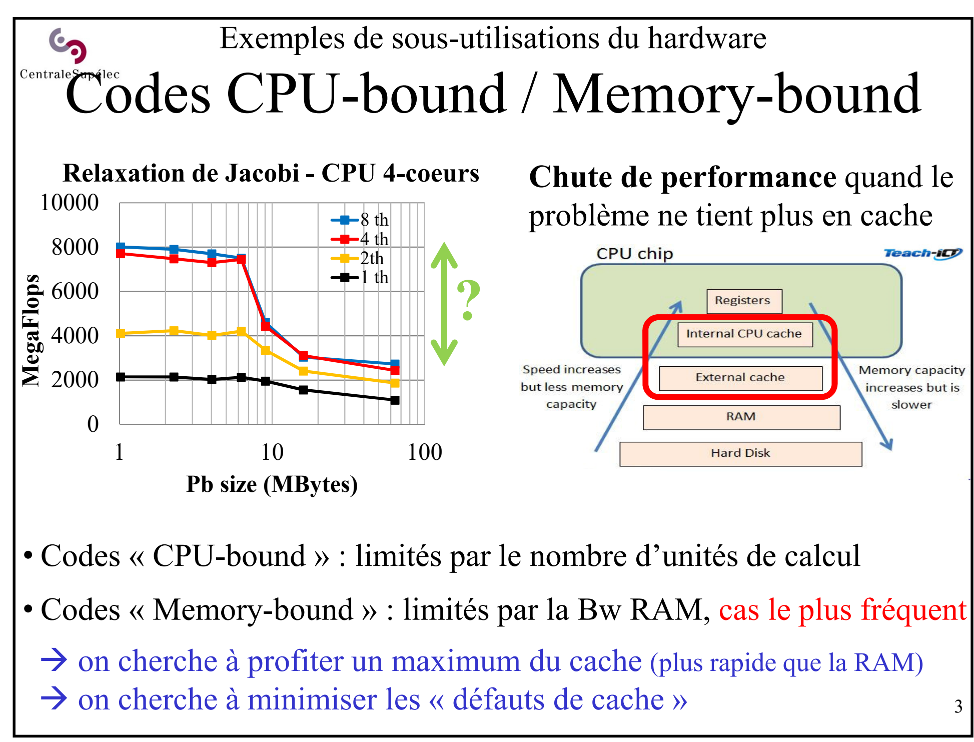
# Risques de sous-utilisations du hardware

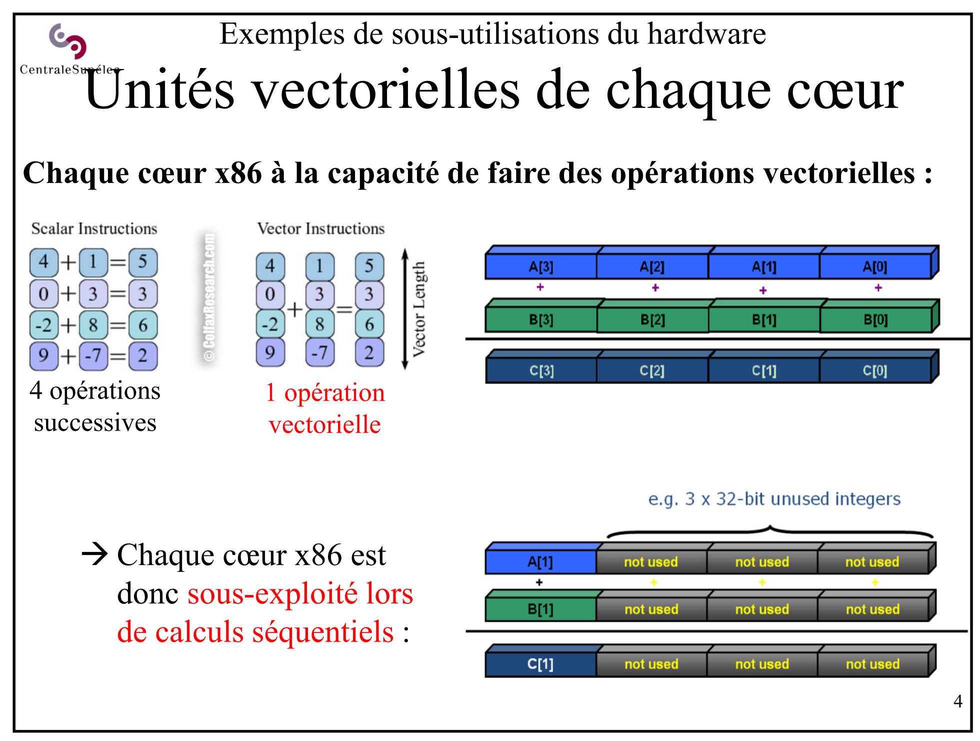
*Bonne ou mauvaise utilisation du cache ?*

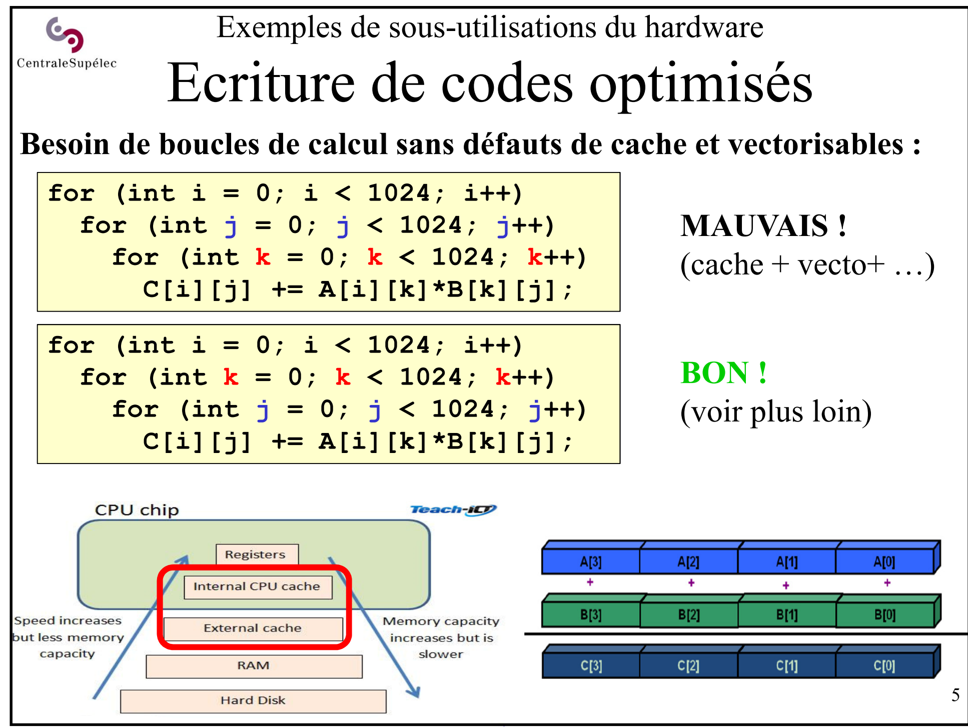



*Bonne ou mauvaise utilisation des unités vectorielles ?*

2


Exemples de sous-utilisations du hardware
CentraleSupélec
Codes CPU-bound / Memory-bound
Relaxation de Jacobi - CPU 4-coeurs
MegaFlops
Pb size (MBytes)
8 th
4 th
2th
1 th
?
Chute de performance quand le problème ne tient plus en cache
CPU chip
Registers
Internal CPU cache
External cache
RAM
Hard Disk
Speed increases but less memory capacity
Memory capacity increases but is slower
• Codes « CPU-bound » : limités par le nombre d'unités de calcul
• Codes « Memory-bound » : limités par la Bw RAM, cas le plus fréquent
→ on cherche à profiter un maximum du cache (plus rapide que la RAM)
→ on cherche à minimiser les « défauts de cache »
3


Exemples de sous-utilisations du hardware
CentraleSupélec
Unités vectorielles de chaque cœur
Chaque cœur x86 à la capacité de faire des opérations vectorielles :
Scalar Instructions
Vector Instructions
Vector Length
4 opérations successives
1 opération vectorielle
A[3] A[2] A[1] A[0]
B[3] B[2] B[1] B[0]
C[3] C[2] C[1] C[0]
e.g. 3 x 32-bit unused integers
not used
→ Chaque cœur x86 est donc sous-exploité lors de calculs séquentiels :
4






CentraleSupélec

# Quelques optimisations possibles

**Minimiser les défauts de cache (les accès à la RAM)**

→ Tirer partie au maximum de la mémoire cache

(blocage en cache/tuilage/*tilling*, ordre des nids de boucles, organisation des données en mémoire…)

**Engager les unités vectorielles de chaque cœur (SSE/AVX units)**

→ Ecrire des boucles internes vectorisables

(éviter les écritures concurrentes dans la même variable, éviter les recouvrements de tableaux (*aliasing*), aligner les accès aux données…

**Minimiser les ruptures de pipeline**

→Déroulement des boucles/*Loop unrolling*

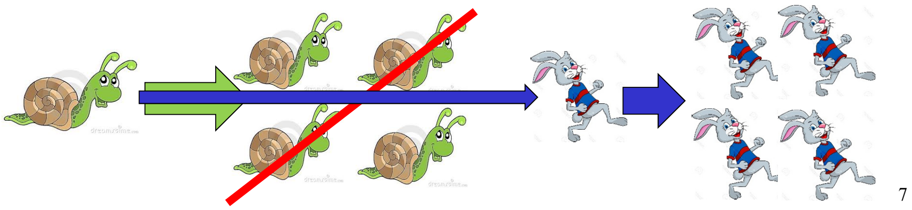
……

# Enjeux des optimisations sérielles et de la vectorisation

Paralléliser un code lent/inefficace est facile :

- les surcoûts de parallélisation ne se voient presque pas !
- la vitesse restera faible, mais l'accélération sera très bonne !

→ Avant de paralléliser, on optimise les « noyaux de calculs »
→ Optimisation maximale sur un cœur







# Evolution des perfs. mono-cœur

Quand on cherche la performance (HPC) on commence par optimiser les noyaux de calculs séquentiels :

Ex : Produit de matrices denses « code TP 3A »

| | Sparc-1 – 1989 | | P3 – 2002 | | Quadcore – 2008 gcc de 2012 | |
|---|---|---|---|---|---|---|
| | Mflops | SU | Mflops | SU | Mflops | SU |
| Séq. basique | 0.31 | *1.00* | 20.00 | *1.00* | 85.00 ? | *1.00* |
| Séq. un peu optimisé | 1.07 | *3.45* | 119.34 | *5.97* | 1881.00 | *22.12* |
| Séq. optimisé (lib) | | | | | 9800.00 | *115.29* |

Et en 2015 avec des processeurs de 2012 →

# Evolution des perfs. mono-cœur

Quand on cherche la performance (HPC) on commence par optimiser les noyaux de calculs séquentiels :

Ex : Produit de matrices denses « code TP 3A »

| | Sparc-1 – 1989 | | P3 – 2002 | | Hexacore – 2012 gcc de 2015 | |
|---|---|---|---|---|---|---|
| | Mflops | SU | Mflops | SU | Mflops | SU |
| Séq. basique | 0.31 | *1.00* | 20.00 | *1.00* | 400.00 | *1.00* |
| Séq. un peu optimisé | 1.07 | *3.45* | 119.34 | *5.97* | 3140.00 | *7.85* |
| Séq. optimisé (lib) | | | | | 24420.00 | *61.05* |

- L'écart de performance entre un programme séquentiel « basique » et un programme séquentiel un minimum optimisé croît.
- ×8 sur un code maison (un peu) optimisé
  ×61 en utilisant des bibliothèques (très) optimisées !





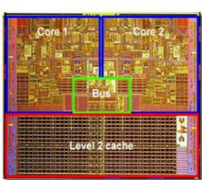
# Besoin de codes optimisés

Les améliorations du hard et des compilateurs ne suffisent pas :
**besoin d'implanter un code optimisé sur chaque cœur**

**Pour aller chercher la performance :**

- Ne pensez pas à l'architecture Von Neuman ni à la machine de Turing !

  Pensez à **l'architecture réelle** de la famille de processeurs que vous utilisez




- Etudiez ce que fait, et ne fait pas, **votre compilateur**
- Apprenez les règles **d'optimisation sérielle** classiques

# Optimisation explicite du code source

Options de compilation

Optimisation du code source

Encouragement de la vectorisation automatique

} Atteindre la vitesse maximale sur un cœur, avec le hardware disponible

11

Optimisation explicite du code source

# Options de compilation

**Tout compilateur propose des « options d'optimisation » :**

- en espace mémoire
- en vitesse de code

Sous Linux : **gcc/icc** **-O0** …. : pas d'optimisation
**-O1** …. : premier niveau normalisé
**-O2** …. : deuxième niveau normalisé
**-O3** …. : troisième niveau normalisé

Ainsi que des optimisations spécifiques :
**gcc –funroll-loops** …. : déroulement forcée des boucles
(voir plus loin)

Sous Windows : d'autres choix existent
(dans des menus de configuration de Visual Studio par exemple)

12

# Options de compilation

**La compilation optimisée :**

- dure plus longtemps
  (surtout avec de la vectorisation)
- est plus exigeante avec la qualité du code source
  (un code peu fonctionner en –O0, et *planter* en –O3)
- mais produit un code beaucoup plus rapide !

→ TOUJOURS enclencher les optimisations du compilateur
(première étape d'une démarche d'optimisation)

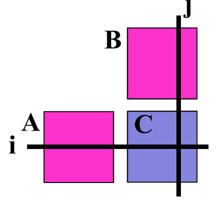
→ ETUDIER ce que propose son compilateur
(lire la documentation/les pages de manuel)

# Problèmes du code src naïf

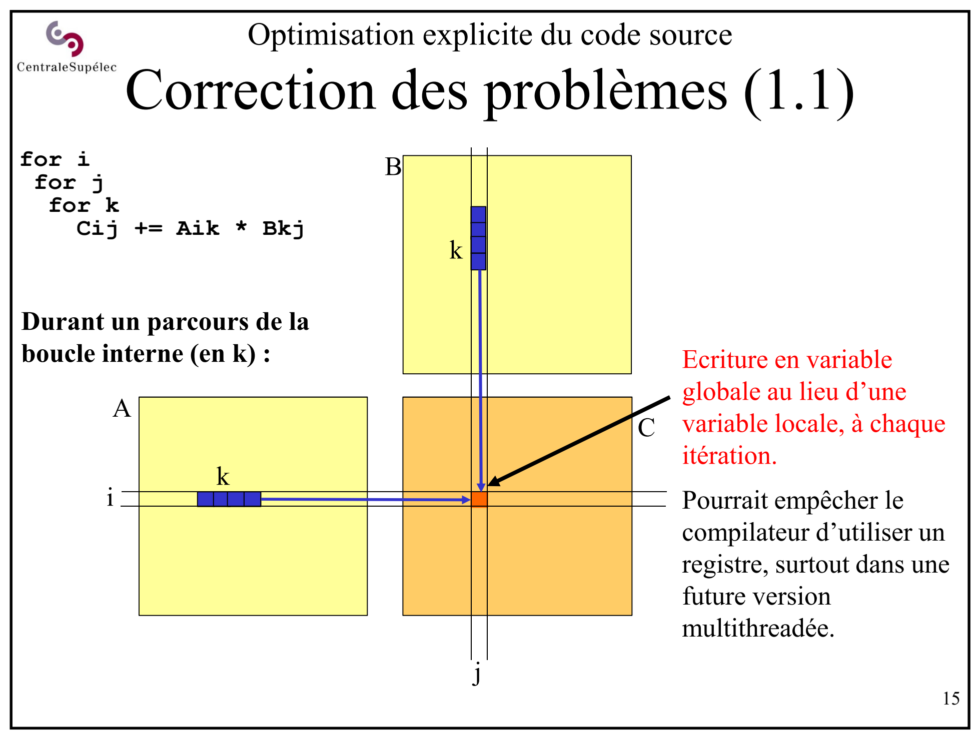
**Version naïve du produit de matrices denses :**

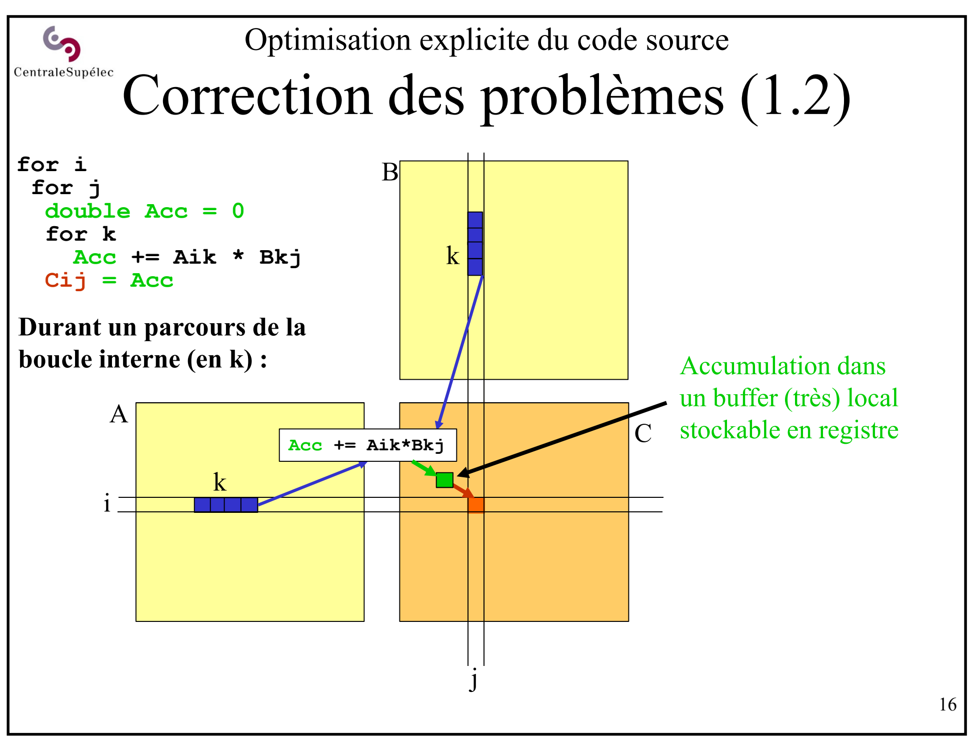
Implantation proche des équations mathématiques (facile à maintenir)

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      C[i][j] += A[i][k]*B[k][j];
```




→ Mauvais usage de la mémoire cache, fréquents « défauts de cache »
→ « Réduction » dans la boucle interne : vectorisation pauvre
→ Si « n » est une variable : vectorisation +difficile pour le compilateur
→ Petit corps de boucle : mauvais usage du pipeline (bcp de « ruptures de pipeline »)
→ Ecriture en var. globale à chaque itération, plutôt qu'en registre local (peut bloquer des optimisations, surtout en multi-threads)


Optimisation explicite du code source
CentraleSupélec
Correction des problèmes (1.1)
for i
 for j
  for k
    Cij += Aik * Bkj
Durant un parcours de la boucle interne (en k) :
B
k
A
k
i
C
j
Ecriture en variable globale au lieu d'une variable locale, à chaque itération.
Pourrait empêcher le compilateur d'utiliser un registre, surtout dans une future version multithreadée.
15


Optimisation explicite du code source
CentraleSupélec
Correction des problèmes (1.2)
for i
 for j
  double Acc = 0
  for k
    Acc += Aik * Bkj
  Cij = Acc
Durant un parcours de la boucle interne (en k) :
B
k
A
Acc += Aik*Bkj
k
i
C
j
Accumulation dans un buffer (très) local stockable en registre
16

Optimisation explicite du code source

CentraleSupélec

# Correction des problèmes (1.3)

**1ère version optimisée :**

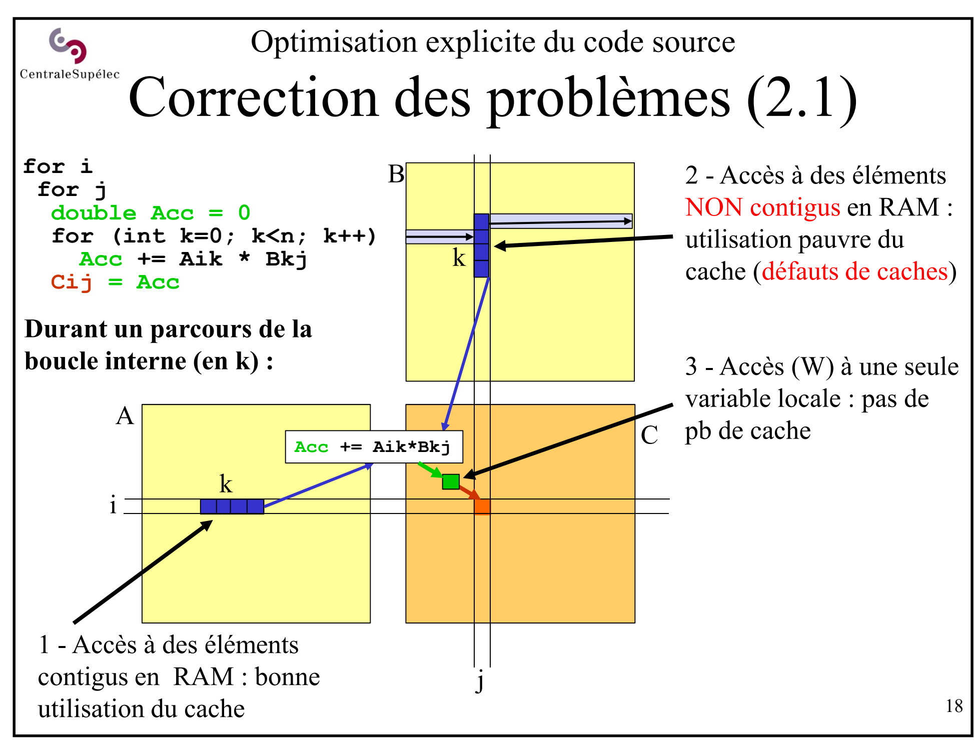
Implantation plus proche de l'architecture du CPU :

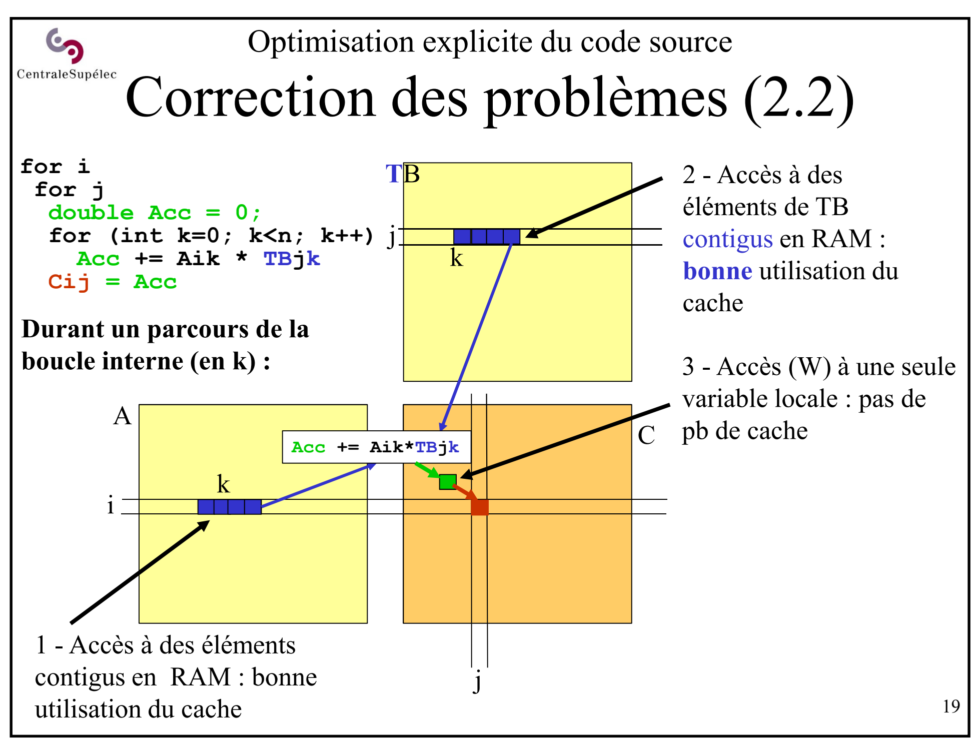
```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    double accu = 0.0;
    for (int k = 0; k < n; k++) {
      accu += A[i][k]*B[k][j];
    }
    C[i][j] = accu;
  }
```

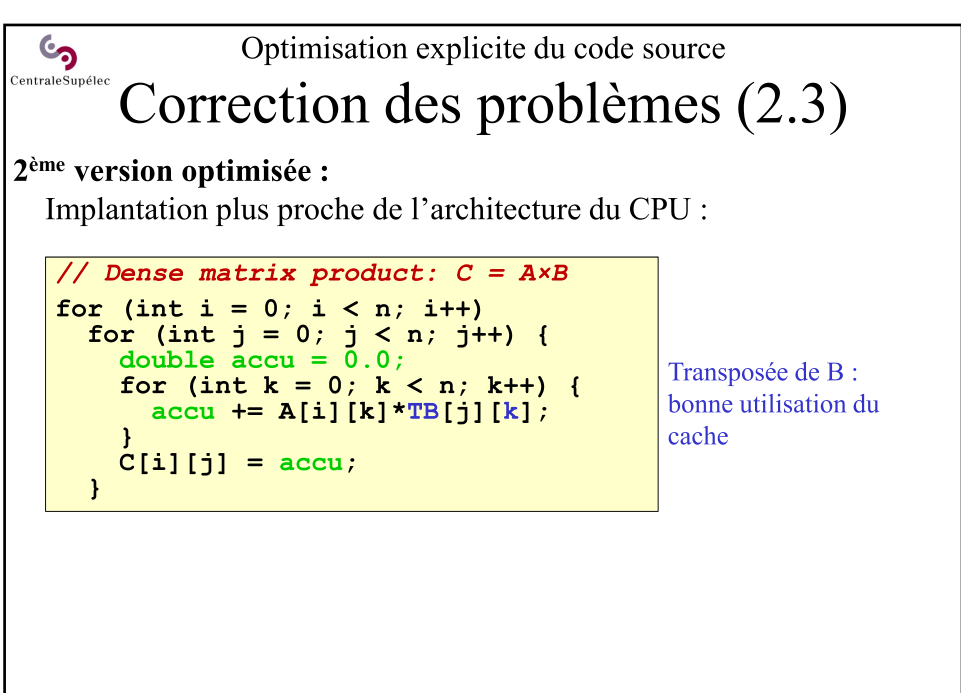
Accumulateur **local:**

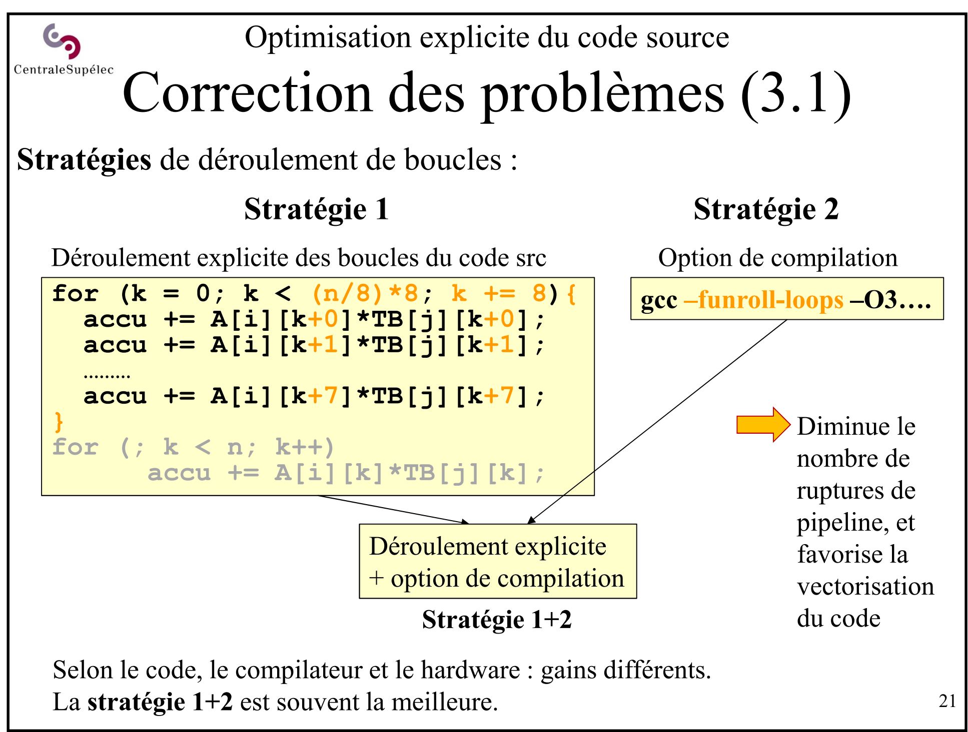
- déclaré localement avec une portée limitée
- non accessible depuis un autre thread (voir suite du cours)
- conservé en registre proche de l'unité de calcul

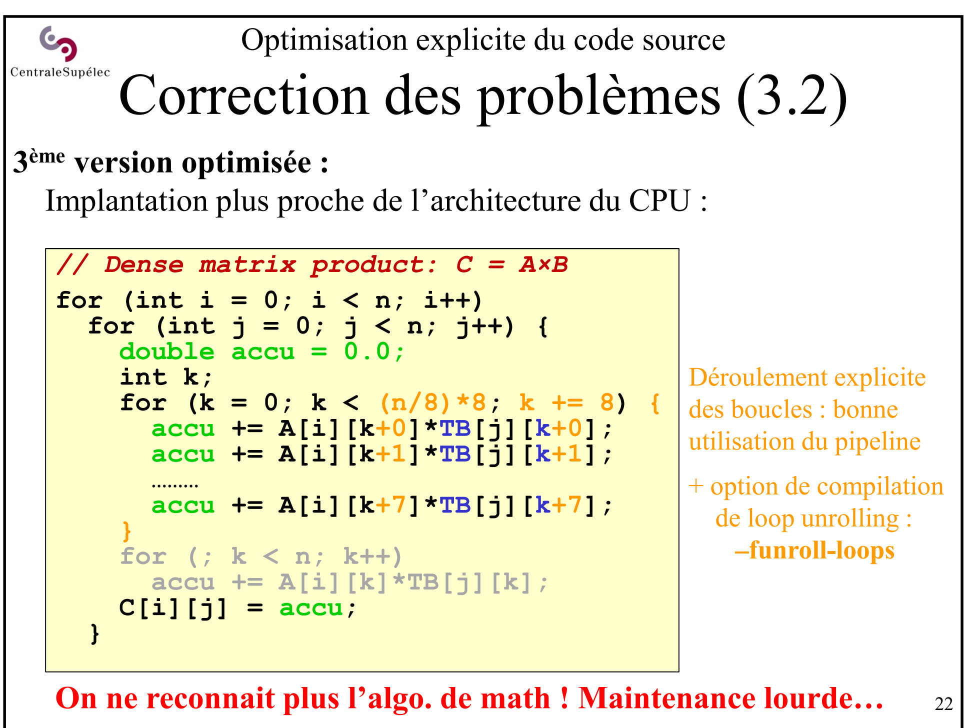
17

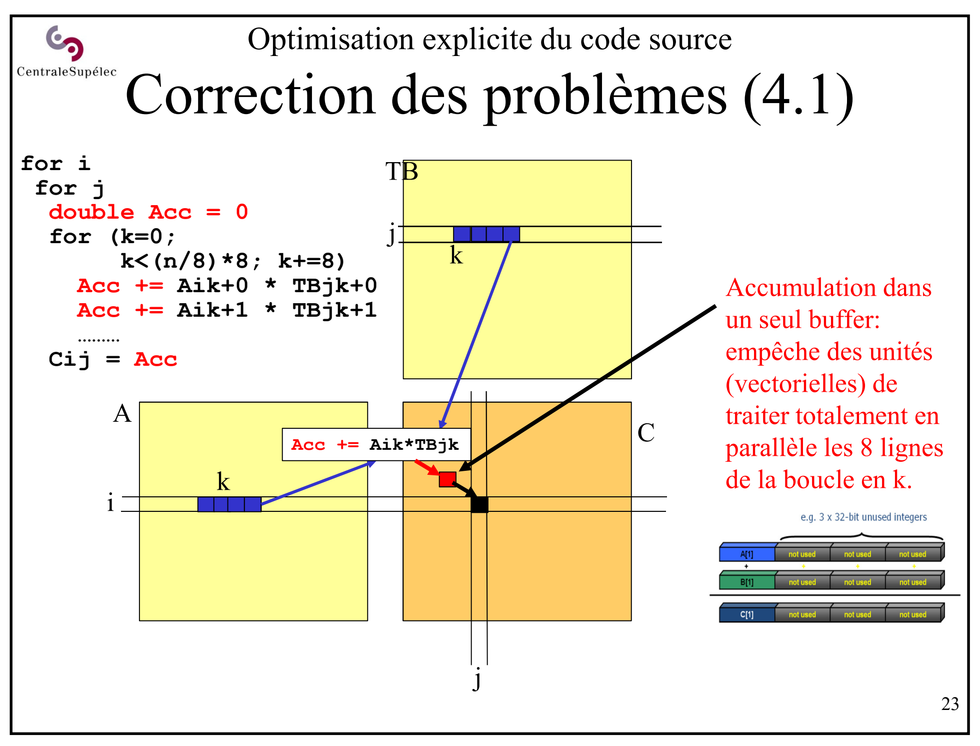


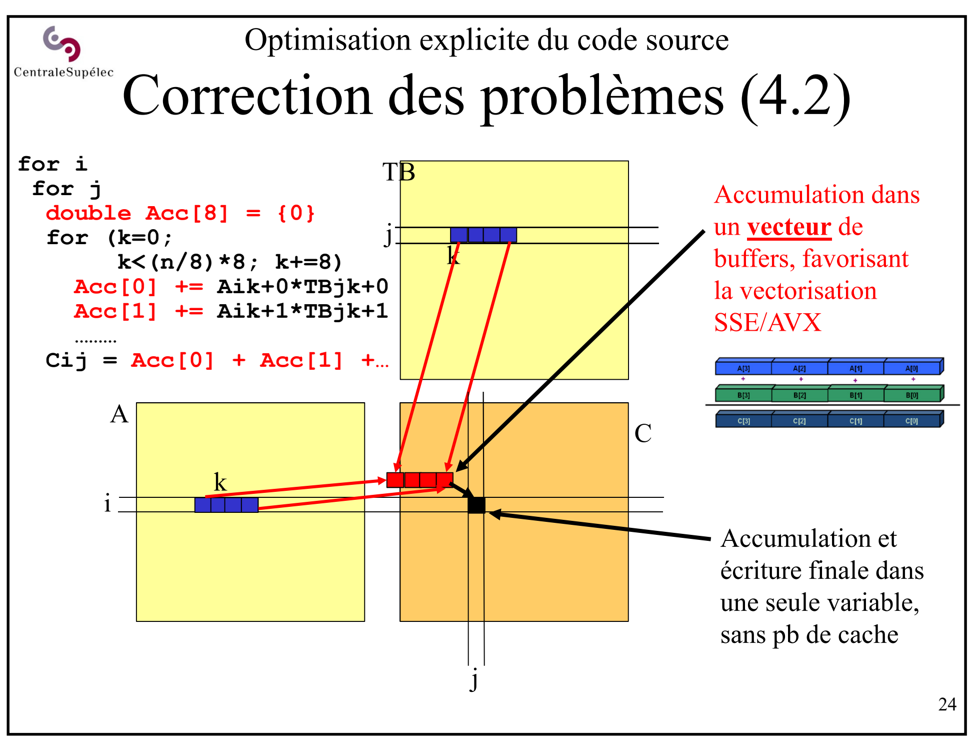

CentraleSupélec
Optimisation explicite du code source
Correction des problèmes (2.2)
for i
for j
double Acc = 0;
for (int k=0; k<n; k++)
Acc += Aik * TBjk
Cij = Acc
Durant un parcours de la boucle interne (en k) :
TB
j
k
2 - Accès à des éléments de TB contigus en RAM : bonne utilisation du cache
3 - Accès (W) à une seule variable locale : pas de pb de cache
A
k
i
Acc += Aik*TBjk
C
j
1 - Accès à des éléments contigus en RAM : bonne utilisation du cache
19


CentraleSupélec
Optimisation explicite du code source
Correction des problèmes (2.3)
2ème version optimisée :
Implantation plus proche de l'architecture du CPU :
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++) {
double accu = 0.0;
for (int k = 0; k < n; k++) {
accu += A[i][k]*TB[j][k];
}
C[i][j] = accu;
}
Transposée de B : bonne utilisation du cache
On reconnait moins l'algo. de math…
20


Optimisation explicite du code source
CentraleSupélec
Correction des problèmes (3.1)
Stratégies de déroulement de boucles :
Stratégie 1
Déroulement explicite des boucles du code src
for (k = 0; k < (n/8)*8; k += 8){
accu += A[i][k+0]*TB[j][k+0];
accu += A[i][k+1]*TB[j][k+1];
.........
accu += A[i][k+7]*TB[j][k+7];
}
for (; k < n; k++)
accu += A[i][k]*TB[j][k];
Stratégie 2
Option de compilation
gcc –funroll-loops –O3….
Diminue le nombre de ruptures de pipeline, et favorise la vectorisation du code
Déroulement explicite
+ option de compilation
Stratégie 1+2
Selon le code, le compilateur et le hardware : gains différents.
La stratégie 1+2 est souvent la meilleure.
21


Optimisation explicite du code source
CentraleSupélec
Correction des problèmes (3.2)
3ème version optimisée :
Implantation plus proche de l'architecture du CPU :
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++) {
double accu = 0.0;
int k;
for (k = 0; k < (n/8)*8; k += 8) {
accu += A[i][k+0]*TB[j][k+0];
accu += A[i][k+1]*TB[j][k+1];
.........
accu += A[i][k+7]*TB[j][k+7];
}
for (; k < n; k++)
accu += A[i][k]*TB[j][k];
C[i][j] = accu;
}
Déroulement explicite des boucles : bonne utilisation du pipeline
+ option de compilation de loop unrolling :
–funroll-loops
On ne reconnait plus l'algo. de math ! Maintenance lourde…
22


CentraleSupélec
Optimisation explicite du code source
Correction des problèmes (4.1)
for i
for j
double Acc = 0
for (k=0;
k<(n/8)*8; k+=8)
Acc += Aik+0 * TBjk+0
Acc += Aik+1 * TBjk+1
.........
Cij = Acc
TB
j
k
A
k
i
Acc += Aik*TBjk
C
j
Accumulation dans un seul buffer: empêche des unités (vectorielles) de traiter totalement en parallèle les 8 lignes de la boucle en k.
23


CentraleSupélec
Optimisation explicite du code source
Correction des problèmes (4.2)
for i
for j
double Acc[8] = {0}
for (k=0;
k<(n/8)*8; k+=8)
Acc[0] += Aik+0*TBjk+0
Acc[1] += Aik+1*TBjk+1
.........
Cij = Acc[0] + Acc[1] +...
TB
j
k
A
k
i
C
j
Accumulation dans un vecteur de buffers, favorisant la vectorisation SSE/AVX
Accumulation et écriture finale dans une seule variable, sans pb de cache
24

CentraleSupélec

# Correction des problèmes (4.3)

**4ème version optimisée :**

Implantation encore plus proche de l'architecture du CPU :

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    double accu[8] = {0.0};
    int k;
    for (k = 0; k < (n/8)*8; k += 8) {
      accu[0] += A[i][k+0]*TB[j][k+0];
      accu[1] += A[i][k+1]*TB[j][k+1];
      .........
      accu[7] += A[i][k+7]*TB[j][k+7];
    }
    for (; k < n; k++)
      accu[0] += A[i][k]*TB[j][k];
    C[i][j] = accu[0] + … + accu[7];
  }
```

Utilisation d'un **vecteur d'accumulateurs :**

- pas de réduction dans une unique variable
- vectorisation automatique possible

- -funroll-loops ???



Optimisation explicite du code source

CentraleSupélec

# Correction des problèmes (4.4)

**4ème version optimisée - BIS :**

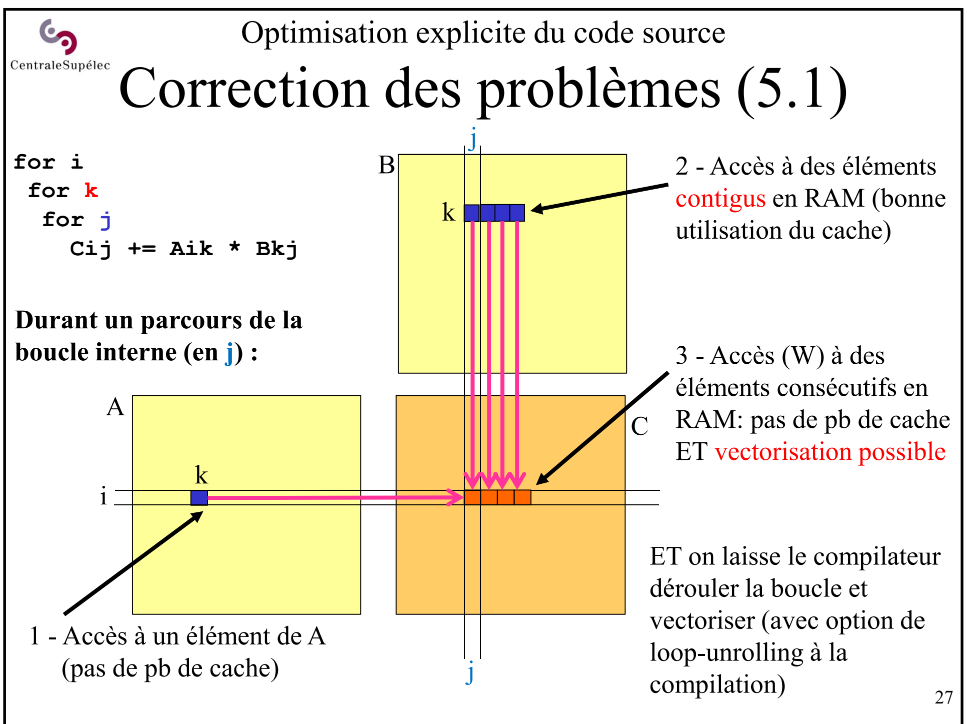
Implantation permettant une vectorisation « intra-instruction » :

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
    double accu = 0;
    int k;
    for (k = 0; k < (n/8)*8; k += 8) {
      accu += A[i][k+0]*TB[j][k+0] +
              A[i][k+1]*TB[j][k+1] +
              .........
              A[i][k+7]*TB[j][k+7];
    }
    for (; k < n; k++)
      accu += A[i][k]*TB[j][k];
    C[i][j] = accu;
  }
```

Utilisation d'un seul accumulateur ET **d'une seule « grosse instruction »** de calcul.

- le compilateur organise ses calculs
- il utilise les registres tampons nécessaires
- donne aussi de bons résultats

- -funroll-loops ???



Optimisation explicite du code source

# Correction des problèmes (5.2)

**5ème Version optimisée par interversion de boucles :**

Implantation proche de l'architecture d'un CPU :

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++)
    for (int j = 0; j < n; j++)
      C[i][j] += A[i][k]*B[k][j];
```

-funroll-loops ??

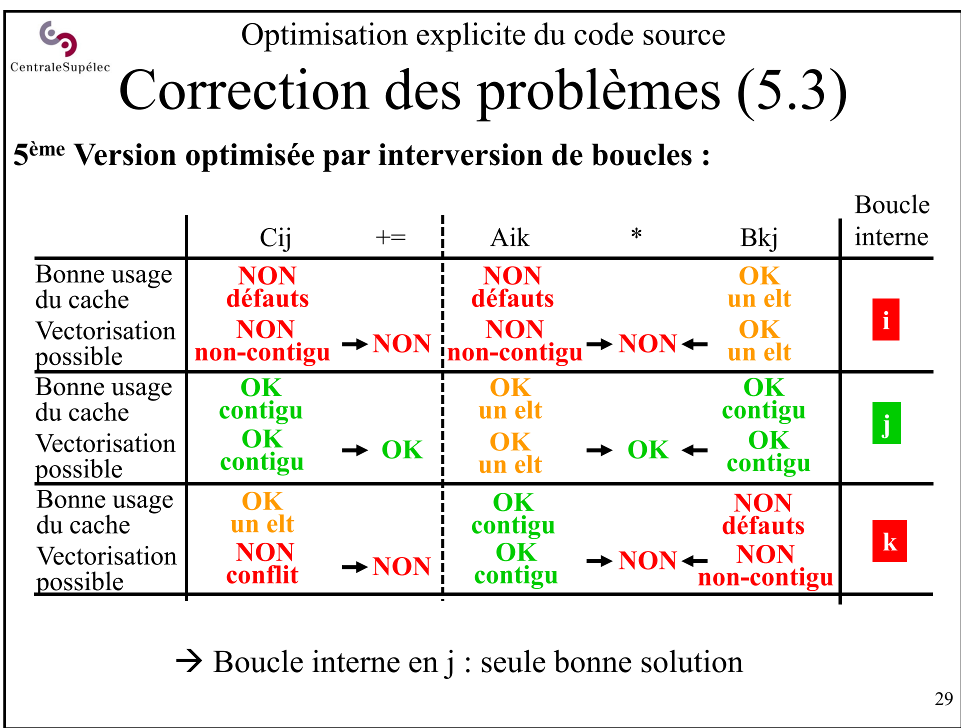
→ meilleure utilisation du cache
- parcours contigu des tableaux dans la boucle interne

→ Suppression de la réduction dans la boucle interne :
- écriture dans des variables différentes et contigues

→ Vectorisation automatique enclenchée par le compilateur

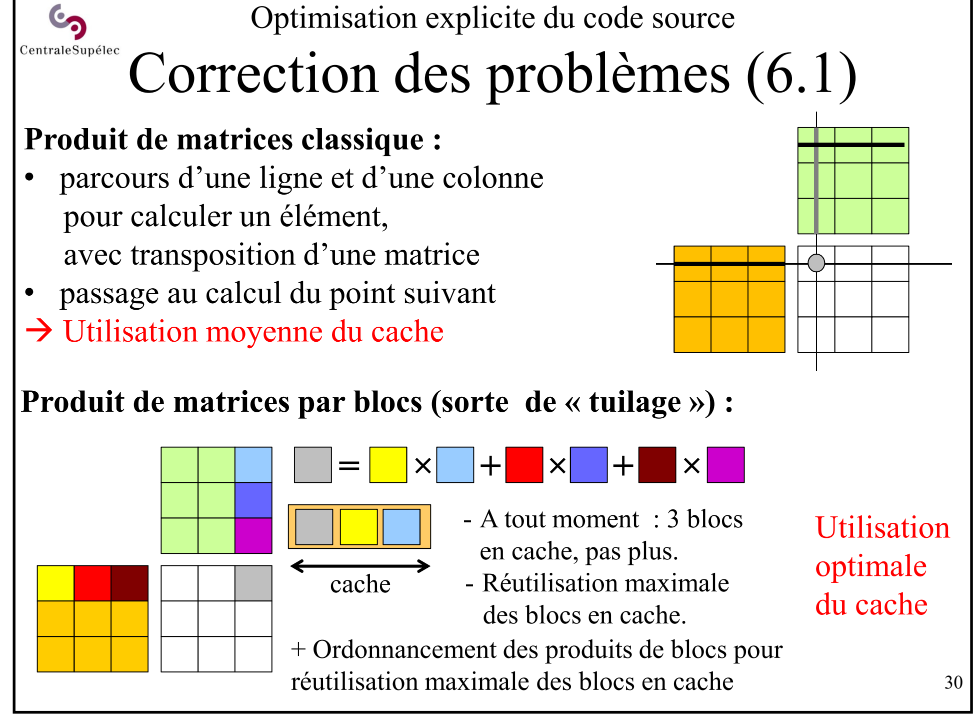
**Pas le plus efficace (pas de blocage en cache), mais simple !**

28




| | Cij | += | Aik | * | Bkj | Boucle interne |
|---|---|---|---|---|---|---|
| Bonne usage du cache | NON défauts | | NON défauts | | OK un elt | i |
| Vectorisation possible | NON non-contigu | → NON | NON non-contigu | → NON ← | OK un elt | |
| Bonne usage du cache | OK contigu | | OK un elt | | OK contigu | j |
| Vectorisation possible | OK contigu | → OK | OK un elt | → OK ← | OK contigu | |
| Bonne usage du cache | OK un elt | | OK contigu | | NON défauts | k |
| Vectorisation possible | NON conflit | → NON | OK contigu | → NON ← | NON non-contigu | |





Optimisation explicite du code source

# Correction des problèmes (6.2)

**Version optimisée AVEC blocage/tuilage (code simplifié) :**

**Hyp simplificatrice :** n = p.Tsize

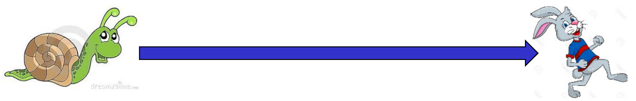
```
// Pour chaque tuile de C :
for (int ii = 0; ii < n; ii += Tsize)
  for (int jj = 0; jj < n; jj += Tsize)
    // - faire une suite de produits de tuiles de A et B
    for (int kk = 0; kk < n; kk += Tsize) {
      // - une tuile A x une tuile B
      for (int i = 0; i < Tsize; i++) {
        for (int j = 0; j < Tsize; j++) {
          double accu[8] = {0.0};
          for (int k = 0; k < Tsize; k += 8) {
            accu[0] += A[ii+i][kk+k+0]*TB[jj+j][kk+k+0];
            accu[1] += A[ii+i][kk+k+1]*TB[jj+j][kk+k+1];
            .........
            accu[7] += A[ii+i][kk+k+7]*TB[jj+j][kk+k+7];
          }
          C[ii+i][jj+j] += accu[0] + … + accu[7];
        }
      }
    }
```

**1 produit de 2 tuiles avec : Tsize = q.8**

31

# Directives de vectorisation

Guidage du compilateur pour la vectorisation



32

# Vectorisation par directives (1)

**Version vectorisée « bas niveau » :**

Point de départ : Implantation proche de l'architecture d'un CPU

```
// Dense matrix product: C = A×B
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++)
    #pragma vector
    #pragma ivdep
    for (int j = 0; j < n; j++)
      C[i][j] += A[i][k]*B[k][j];
```

ICC
GCC ??
ou par
OpenMP

→ Meilleure utilisation du cache
- parcours des tableaux dans l'ordre dans la boucle interne

→ Suppression de la réduction dans la boucle interne :
- écriture dans des variables différentes et contigues

→ Vectorisation guidée **de la boucle interne**, par des directives simples (rôle précis, bien maîtrisable)





# Vectorisation par directives (2)

**Version vectorisée « haut niveau » :**

Point de départ : Implantation proche de l'architecture d'un CPU

```
// Dense matrix product: C = A×B
#pragma simd
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++)
    for (int j = 0; j < n; j++)
      C[i][j] += A[i][k]*B[k][j];
```

ICC
GCC ??
ou par
OpenMP

→ Meilleure utilisation du cache
- parcours des tableaux dans l'ordre dans la boucle interne

→ Suppression de la réduction dans la boucle interne :
- écriture dans des variables différentes et contigues

**Avis diver-geants**

→ Vectorisation guidée **globale**, par une directive de haut niveau sur la boucle externe (fonctionnement moins prévisible)

# Aperçu des performances obtenues

# TP d'optim. sérielles et vectorisation

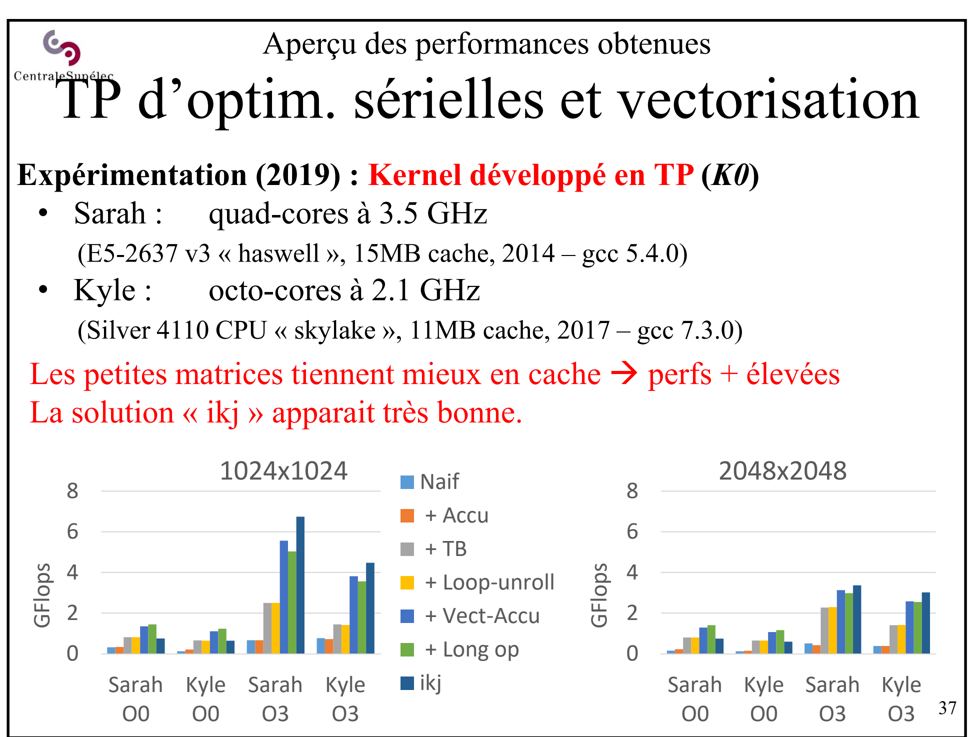
**Expérimentation (2018) :**

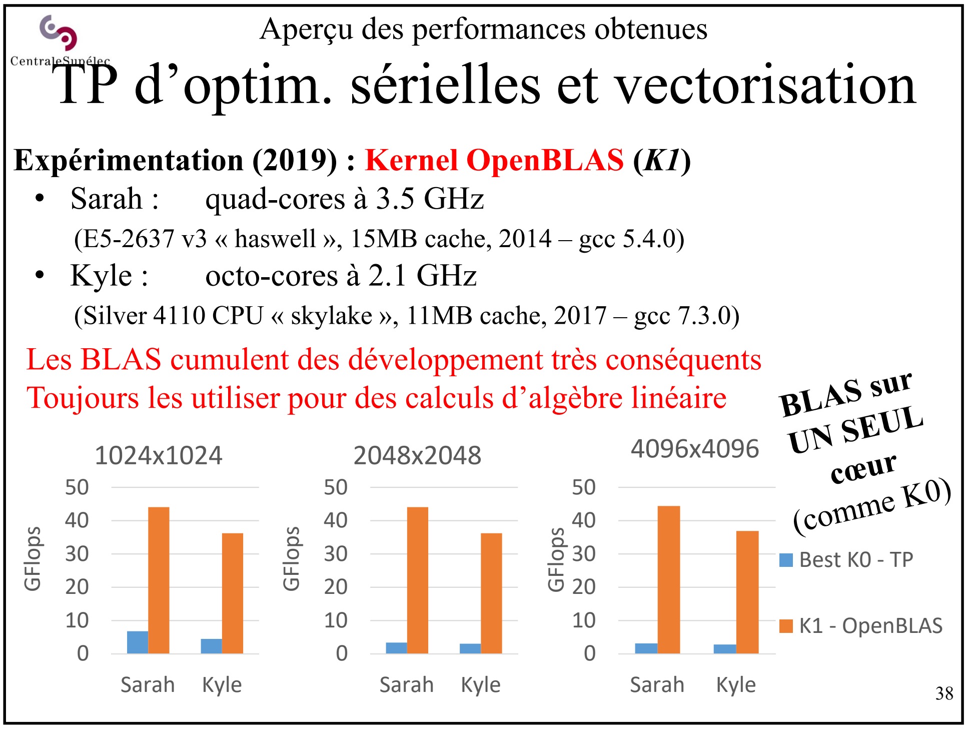
- produit de matrices denses 4096x4096 double
- processeur Intel Xeon *Haswell* E5-2637 v3 - 2014

(4 cœurs physiques – 2 threads/cœur)

| Seq. Naif + O0 | Seq. Naif + O3 | Vectorisé + optim + O3 (TP 3A) | BLAS monothread (OpenBLAS) |
|---|---|---|---|
| 0.12 Gflops | 0.35 Gflops | 3.10 Gflops | 46.3 Gflops |
| x1.0 | x2.9 | x25.8 | x385.8 |

→ Il faut utiliser les bibliothèques optimisées quand c'est possible
→ Quand ce n'est pas possible il faut savoir optimiser à la main


Aperçu des performances obtenues
CentraleSupélec
TP d'optim. sérielles et vectorisation
Expérimentation (2019) : Kernel développé en TP (K0)
• Sarah : quad-cores à 3.5 GHz
(E5-2637 v3 « haswell », 15MB cache, 2014 – gcc 5.4.0)
• Kyle : octo-cores à 2.1 GHz
(Silver 4110 CPU « skylake », 11MB cache, 2017 – gcc 7.3.0)
Les petites matrices tiennent mieux en cache → perfs + élevées
La solution « ikj » apparait très bonne.
1024x1024
2048x2048
GFlops
8
6
4
2
0
Sarah O0
Kyle O0
Sarah O3
Kyle O3
Naif
+ Accu
+ TB
+ Loop-unroll
+ Vect-Accu
+ Long op
ikj
37


Aperçu des performances obtenues
CentraleSupélec
TP d'optim. sérielles et vectorisation
Expérimentation (2019) : Kernel OpenBLAS (K1)
• Sarah : quad-cores à 3.5 GHz
(E5-2637 v3 « haswell », 15MB cache, 2014 – gcc 5.4.0)
• Kyle : octo-cores à 2.1 GHz
(Silver 4110 CPU « skylake », 11MB cache, 2017 – gcc 7.3.0)
Les BLAS cumulent des développement très conséquents
Toujours les utiliser pour des calculs d'algèbre linéaire
BLAS sur UN SEUL cœur (comme K0)
1024x1024
2048x2048
4096x4096
GFlops
50
40
30
20
10
0
Sarah
Kyle
Best K0 - TP
K1 - OpenBLAS
38

## Optimisations sérielles et vectorisation

# Questions ?

39