CentraleSupélec

Mineure HPC-SBD

Optimisations sérielles et vectorisation

Stéphane Vialle

CentraleSupélec université PARIS-SACLAY — École doctorale Sciences et technologies de l'information et de la communication (STIC) — RISEGrid — Grand Est

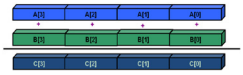
Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

Risques de sous-utilisations du hardware

Bonne ou mauvaise utilisation du cache ?

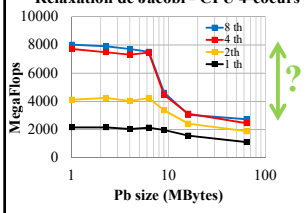
Bonne ou mauvaise utilisation des unités vectorielles ?



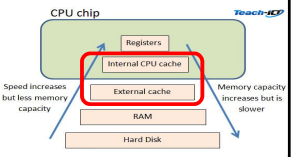
Exemples de sous-utilisations du hardware

Codes CPU-bound / Memory-bound

Relaxation de Jacobi - CPU 4-coeurs



Chute de performance quand le problème ne tient plus en cache



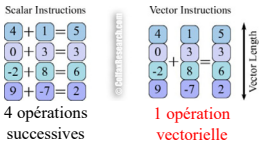
- Codes « CPU-bound » : limités par le nombre d'unités de calcul
- Codes « Memory-bound » : limités par la Bw RAM, cas le plus fréquent

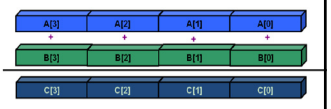
→ on cherche à profiter un maximum du cache (plus rapide que la RAM)
→ on cherche à minimiser les « défauts de cache »

Exemples de sous-utilisations du hardware

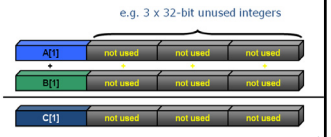
Unités vectorielles de chaque cœur

Chaque cœur x86 à la capacité de faire des opérations vectorielles :





→ Chaque cœur x86 est donc sous-exploité lors de calculs séquentiels :



Exemples de sous-utilisations du hardware

Ecriture de codes optimisés

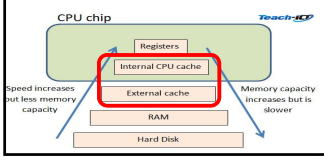
Besoin de boucles de calcul sans défauts de cache et vectorisables :

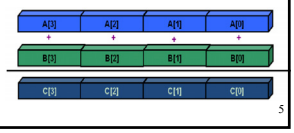
```
for (int i = 0; i < 1024; i++)
  for (int j = 0; j < 1024; j++)
    for (int k = 0; k < 1024; k++)
      C[i][j] += A[i][k]*B[k][j];
```

MAUVAIS !
(cache + vecto+ …)

```
for (int i = 0; i < 1024; i++)
  for (int k = 0; k < 1024; k++)
    for (int j = 0; j < 1024; j++)
      C[i][j] += A[i][k]*B[k][j];
```

BON !
(voir plus loin)





Exemples de sous-utilisations du hardware

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…

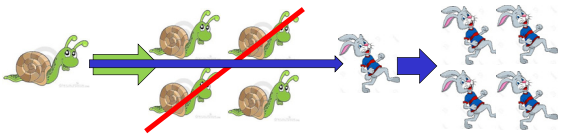
Minimiser les ruptures de pipeline
→Déroulement des boucles/*Loop unrolling*

……

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 !

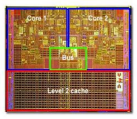
Besoin de codes optimisés

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

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)

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)

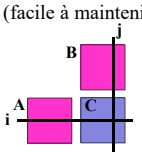
→ ETUDIER ce que propose son compilateur
(lire la documentation/les pages de manuel)

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];
```

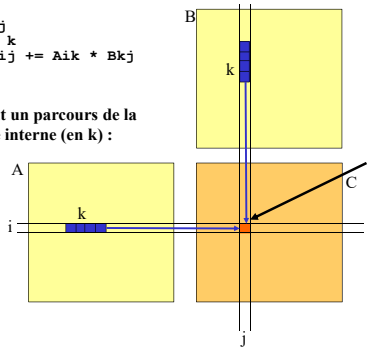


→ Mauvais usage de la mémoire cache, fréquents « défauts de cache »
→ « Réduction » dans la boucle interne : vectorisation pauvre
→ Si « n » est une variable : vectorisation +difficile pour le compilateur
→ Petit corps de boucle : mauvais usage du pipeline (bcp de « ruptures de pipeline »)
→ Ecriture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

Correction des problèmes (1.1)

```
for i
 for j
  for k
    Cij += Aik * Bkj
```

Durant un parcours de la boucle interne (en k) :



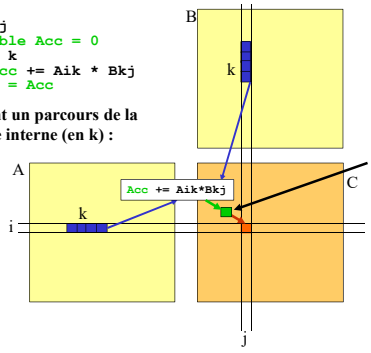
Ecriture en variable globale au lieu d'une variable locale, à chaque itération.

Pourrait empêcher le compilateur d'utiliser un registre, surtout dans une future version multithreadée.

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) :



Accumulation dans un buffer (très) local stockable en registre

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;
  }
```

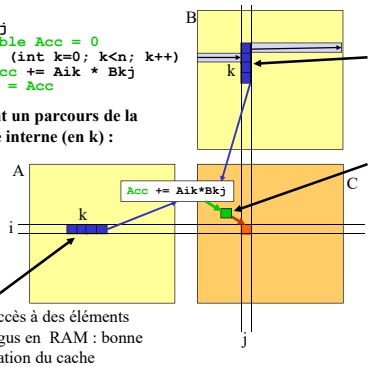
Accumulateur **local:**

- 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

Correction des problèmes (2.1)

```
for i
 for j
  double Acc = 0
  for (int k=0; k<n; k++)
    Acc += Aik * Bkj
  Cij = Acc
```

Durant un parcours de la boucle interne (en k) :



2 - Accès à des éléments NON contigus en RAM : utilisation pauvre du cache (défauts de caches)

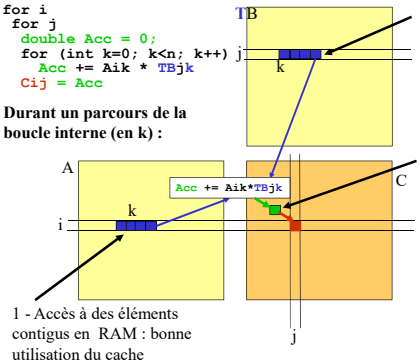
3 - Accès (W) à une seule variable locale : pas de pb de cache

1 - Accès à des éléments contigus en RAM : bonne utilisation du cache

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) :



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

1 - Accès à des éléments contigus en RAM : bonne utilisation du cache

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…

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];
```

Déroulement explicite + option de compilation

Stratégie 1+2

Stratégie 2

Option de compilation

gcc –funroll-loops –O3….

Diminue le nombre de ruptures de pipeline, et favorise la vectorisation du code

Selon le code, le compilateur et le hardware : gains différents.
La **stratégie 1+2** est souvent la meilleure.

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;
  }
```

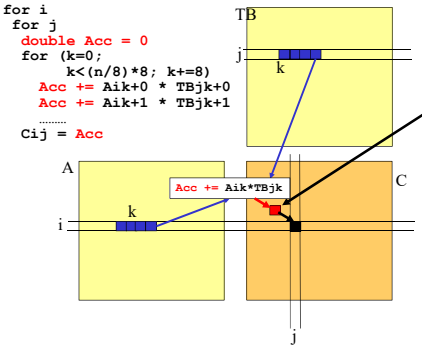
Déroulement explicite des boucles : bonne utilisation du pipeline

\+ option de compilation de loop unrolling : **–funroll-loops**

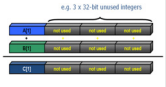
On ne reconnait plus l'algo. de math ! Maintenance lourde…

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
```

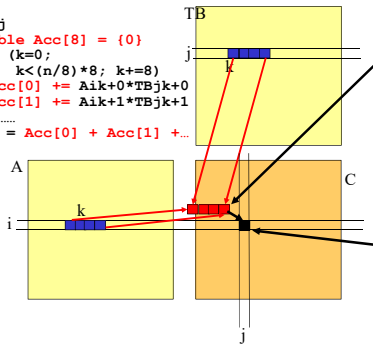


Accumulation dans un seul buffer: empêche des unités (vectorielles) de traiter totalement en parallèle les 8 lignes de la boucle en k.

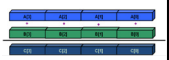


Correction des problèmes (4.2)

```
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] +…
```



Accumulation dans un **vecteur** de buffers, favorisant la vectorisation SSE/AVX



Accumulation et écriture finale dans une seule variable, sans pb de cache

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 ???

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.

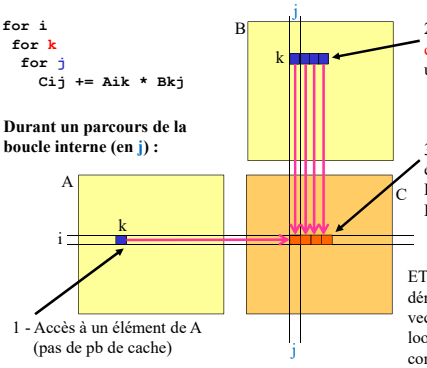
- le compilateur organise ses calculs
- il utilise les registres tampons nécessaires
- donne aussi de bons résultats

- -funroll-loops ???

Correction des problèmes (5.1)

```
for i
 for k
  for j
    Cij += Aik * Bkj
```

Durant un parcours de la boucle interne (en j) :



2 - Accès à des éléments contigus en RAM (bonne utilisation du cache)

3 - Accès (W) à des éléments consécutifs en RAM: pas de pb de cache ET vectorisation possible

1 - Accès à un élément de A (pas de pb de cache)

ET on laisse le compilateur dérouler la boucle et vectoriser (avec option de loop-unrolling à la compilation)

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 !

Correction des problèmes (5.3)

5ème Version optimisée par interversion de boucles :

	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	

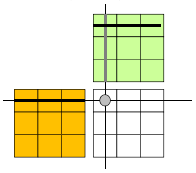
→ Boucle interne en j : seule bonne solution

Correction des problèmes (6.1)

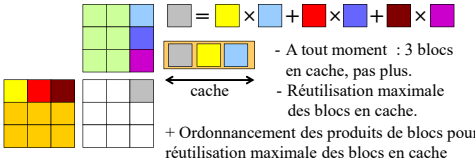
Produit de matrices classique :

- parcours d'une ligne et d'une colonne pour calculer un élément, avec transposition d'une matrice
- passage au calcul du point suivant

→ Utilisation moyenne du cache



Produit de matrices par blocs (sorte de « tuilage ») :



- A tout moment : 3 blocs en cache, pas plus.
- Réutilisation maximale des blocs en cache.

+ Ordonnancement des produits de blocs pour réutilisation maximale des blocs en cache

Utilisation optimale du cache

Correction des problèmes (6.2)

Version optimisée AVEC blocage/tuilage (code simplifié) :

```
// 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];
        }
      }
    }
```

Hyp simplificatrice : n = p.Tsize

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

Directives de vectorisation

Guidage du compilateur pour la vectorisation

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

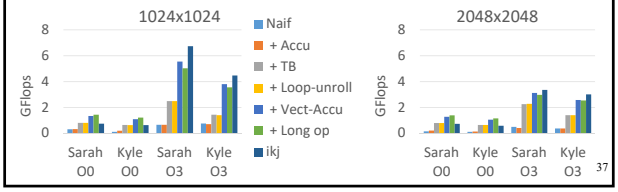
Aperçu des performances obtenues

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.



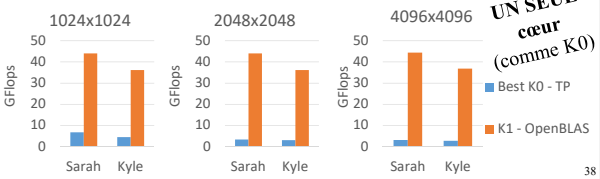
Aperçu des performances obtenues

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)

Optimisations sérielles et vectorisation

Questions ?