Architecture et optimisations mono-cœur CPU

EI-SE5 – Calcul haute performance

Adrien Cassagne

Le 20 septembre 2022

Objectifs de ce cours

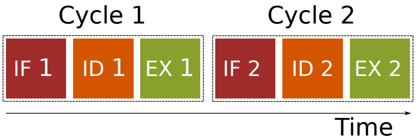
- Comprendre plus finement certains choix architecturaux des CPU actuels
  - Pipeline
  - Super-scalaire
  - Out-of-order
  - Exemples de micro-architectures récentes (Firestorm et Alder Lake)
- Montrer comment des modifications sur le code source, tenant compte de la micro-architecture, peuvent réduire le temps d'exécution
  - Inlining
  - Fusion de boucles
  - Séparation de boucle
  - Déroulage de boucle
  - Rotation de variables
  - Accès mémoires
- Dans ce cours on ne verra **PAS** la programmation SIMD

Table des matières

2 Architecture CPU mono-cœur

Processeur sans pipeline

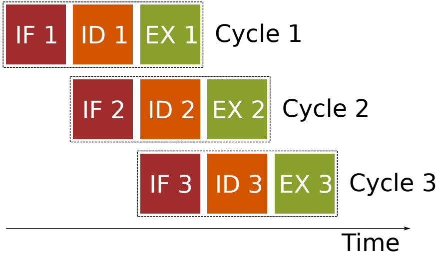
- Généralement on peut décomposer une instruction en plusieurs sous-étapes:
  1. *Fetch* (IF) : l'instruction est copiée depuis la mémoire
  2. *Decode* (ID) : l'instruction est interprétée par le CPU
  3. *Execute* (EX) : l'instruction est exécutée
- Sur un processeur SANS pipeline cela correspond à :



- L'exécution du cycle 1 et du cycle 2 prend 6 unités de temps

Processeur avec pipeline

- D'après ce que nous venons de voir, il est possible de diviser une instruction en 3 sous-instructions (μop)
- Nous considérons que chaque sous-instruction prend le même temps
- Sur un processeur AVEC pipeline cela correspond à :

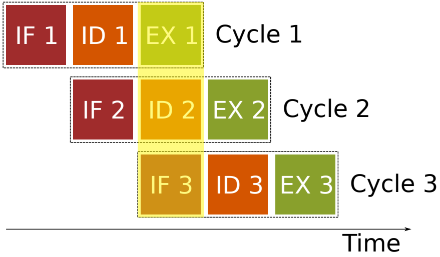


- L'exécution du cycle 1 et du cycle 2 prend 5 unités de temps
- Pipeline à 3 étages (IF, ID et EX peuvent s'exécuter en parallèle)

Processeur avec pipeline

2 Architecture CPU mono-cœur

- Il y a un certain temps avant que le pipeline soit optimal
  - C'est ce que l'on appelle la latence du pipeline (2 cycles ici)
- Si on ne considère pas la latence, un pipeline à 3 étages est 3 fois plus rapide
- En jaune : le moment à partir du quel le pipeline est optimal :

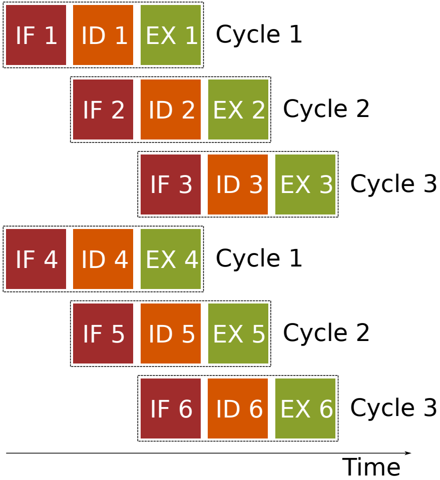


Processeur avec pipeline

- Aujourd'hui les processeurs ont des pipelines de 10 à 20 étages mais le principe reste le même
- Il y a un mécanisme pour que l'instruction précédente puisse directement passer son résultat à l'instruction suivante (*register latch* + *forwarding*)
- La stratégie du pipeline semble efficace mais que se passe t-il si on a des conditions (`if`) ?
  - Dans ce cas c'est problématique, le processeur ne peut pas savoir qu'elle sera la prochaine instruction, cela est néfaste pour l'efficacité du pipeline (cela crée des "bulles")
  - Il existe des mécanismes de prédiction de branchement pour limiter cet effet...

Processeur super-scalaire

- C'est la capacité d'un processeur à exécuter plusieurs instructions en parallèle
  - *Instruction Level Parallelism* (ILP)
  - Les processeurs sont généralement super-scalaires sur 3 à 10 niveaux
- Cela signifie qu'un CPU est capable d'exécuter plus d'une instruction par cycle (3 à 10 instructions)
  - L'exemple à droite montre un processeur avec un ILP de 2 et un pipeline à 3 étages
  - Par exemple, un processeur peut calculer 1 multiplication puis charger des données depuis la mémoire en 1 cycle



Exécution *Out-of-Order* (OoO)

- La plupart des processeurs sont maintenant capables d'exécuter des instructions dans un ordre différent de celui donné par le code assembleur
  - On dit que le processeur peut exécuter des instructions dans le désordre
  - Cela permet de maximiser l'utilisation des unités du CPU pendant un cycle (processeur super-scalaire)
  - Il est difficile de prédire dans quel ordre le processeur va exécuter les instructions
- Voici un exemple pour illustrer :

```c
int a, b, c, d, e, f, g, h;
c = a + b; // 1 - première instruction à être exécutée, elle ne dépend de rien
e = c * d; // 3 - troisième instruction à être exécutée car elle dépend de 'c'
h = f * g; // 2 - deuxième instruction à être exécutée car elle ne dépend de rien
```

Exécution *Out-of-Order* (OoO)

- La plupart des processeurs sont maintenant capables d'exécuter des instructions dans un ordre différent de celui donné par le code assembleur
  - On dit que le processeur peut exécuter des instructions dans le désordre
  - Cela permet de maximiser l'utilisation des unités du CPU pendant un cycle (processeur super-scalaire)
  - Il est difficile de prédire dans quel ordre le processeur va exécuter les instructions
- Voici un exemple pour illustrer :

```c
int a, b, c, d, e, f, g, h;
c = a + b; // 1 - première instruction à être exécutée, elle ne dépend de rien
e = c * d; // 3 - troisième instruction à être exécutée car elle dépend de 'c'
h = f * g; // 2 - deuxième instruction à être exécutée car elle ne dépend de rien
```

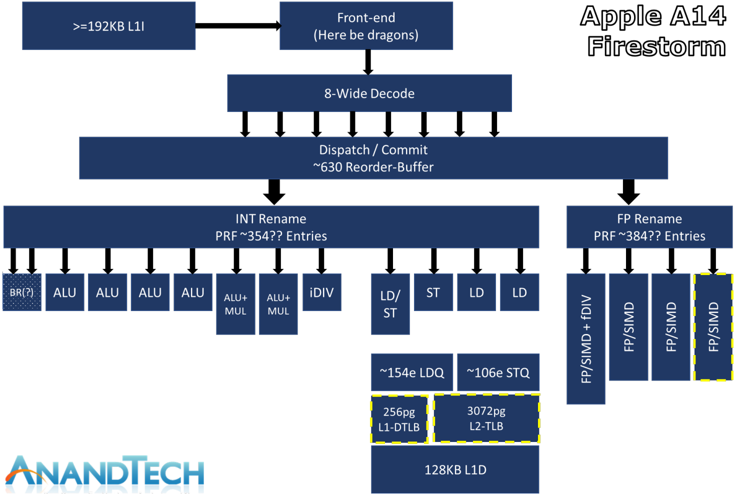
Exécution *Out-of-Order* (OoO)

- La plupart des processeurs sont maintenant capables d'exécuter des instructions dans un ordre différent de celui donné par le code assembleur
  - On dit que le processeur peut exécuter des instructions dans le désordre
  - Cela permet de maximiser l'utilisation des unités du CPU pendant un cycle (processeur super-scalaire)
  - Il est difficile de prédire dans quel ordre le processeur va exécuter les instructions
- Voici un exemple pour illustrer :

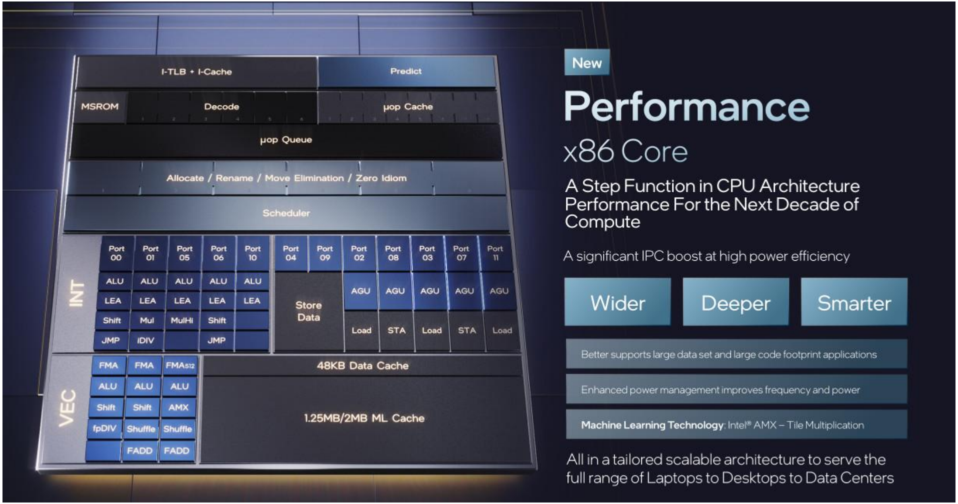
```c
int a, b, c, d, e, f, g, h;
c = a + b; // 1 - première instruction à être exécutée, elle ne dépend de rien
e = c * d; // 3 - troisième instruction à être exécutée car elle dépend de 'c'
h = f * g; // 2 - deuxième instruction à être exécutée car elle ne dépend de rien
```

Micro-architecture Apple Silicon M1 (2020)

2 Architecture CPU mono-cœur



Micro-architecture Intel Alder Lake (2021)



Micro-architecture Intel Alder Lake (2021)

Micro-architecture Intel Alder Lake (2021)

Table des matières

3 Optimisations CPU mono-cœur

Travailler avec le compilateur

- Les compilateurs viennent avec de nombreuses options qui permettent de faire des optimisations automatiquement et ainsi réduire le temps d'exécution de votre programme
- Dans ce cours, nous utiliserons le compilateur C/C++ GNU (`gcc`, `g++`) mais il existe des options similaires chez les autres compilateurs existants
- Il est important de comprendre les optimisations qui peuvent et ne peuvent pas être faites par le compilateur !
  - Cela permet de maximiser la lisibilité du code source tout en conservant une bonne efficacité
  - Le compilateur peut ainsi appliquer des optimisations "sales" à notre place lorsqu'il génère le code binaire

Compilateur : options d'optimisation

- Les options les plus connues sont (`-O[level]`) :
  - `-O0`: pas d'optimisation
  - `-O1`: active une série d'optimisations dans le but de réduire la taille du binaire et son temps d'exécution tout en conservant un temps de compilation relativement faible
  - `-O2`: active toutes les optimisations possibles (exceptées les optimisations qui demandent un compromis entre efficacité et taille du binaire), cette option demande un temps de compilation plus long que `-O1`,
  - `-O3`: optimise encore plus, active les options suivantes : `-finline-functions`, `-funswitch-loops`, `-fpredictive-commoning`, `-fgcse-after-reload`, `-ftree-loop-vectorize`, `-ftree-loop-distribute-patterns`, `-ftree-slp-vectorize`, `-fvect-cost-model`, `-ftree-partial-pre` et `-fipa-cp-clone`.

Compilateur : quelques options spécifiques

- `-finline-functions`: active l'inlining automatique, le compilateur a le choix d'inliner ou non (cette option n'est pas une garantie)
- `-ftree-vectorize`: active la vectorisation automatique du code
- `-ffast-math`: ne tient pas compte de la spécifications IEEE 754 dans les calculs flottants (risque de perte de précision = risque de bugs)
- `-funroll-loops`: déroule les boucles dont les bornes sont connues à la compilation. Cette option fait grossir le code binaire, et n'améliore pas forcément le temps d'exécution
- `-march=native`: active les instructions spécifiques à la micro-architecture sur laquelle le compilateur est exécuté. Souvent nécessaire pour la vectorisation du code
- La documentation des options d'optimisation: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Latence et débit des instructions

- Voici le coût des principales instructions arithmétiques sur micro-architecture Intel Skylake
  - `add`: latence de 4 cycles, débit de 0,5 cycles par instruction
  - `sub`: latence de 4 cycles, débit de 0,5 cycles par instruction
  - `mul`: latence de 4 cycles, débit de 0,5 cycles par instruction
  - `div`: latence de 14 cycles, débit de 4 cycles par instruction
- Comme on peut le voir la division est beaucoup moins efficace que la multiplication (débit 8 fois inférieur)
- Il est donc intéressant de calculer l'inverse et de multiplier par l'inverse ensuite
- Attention tout de même, cette opération induit un perte de précision dans les calculs

Exemple de la division


```c
for (int i = 0; i < n; i++)
    B[i] = A[i] / 3.f;
```

- Nombre de cycles théoriques : $n \times 4$

```c
float inv3 = 1.f / 3.f;
for (int i = 0; i < n; i++)
    B[i] = A[i] * inv3;
```

- Nombre de cycles théoriques : $4 + n \times 0.5$

Fonctions spéciales

- Voici le coût de certaines fonctions mathématiques usuelles :
  - `sqrt`: latence de 12 cycles, débit de 3 cycles par instruction
  - `rsqrt`: latence de 4 cycles, débit de 1 cycle par instruction
  - `pow`: très cher, dépend de l'implémentation, il n'y a pas d'unité matérielle dédiée
  - `cos`: très cher, dépend de l'implémentation, il n'y a pas d'unité matérielle dédiée
  - `sin`: très cher, dépend de l'implémentation, il n'y a pas d'unité matérielle dédiée
  - `tan`: très cher, dépend de l'implémentation, il n'y a pas d'unité matérielle dédiée
- `rsqrt` a un débit d'une instruction par cycle!
  - C'est surprenant mais cette fonction est implémentée en matériel
  - La précision est généralement plus faible que pour les autres instructions
  - Très utilisée pour les calculs de distance 2D/3D
- `pow`, `cos`, `sin` and `tan` sont très coûteuses, il faut essayer de limiter leur utilisation dans les codes HPC
- Il existe souvent des fonctions approchantes qui sont moins coûteuses

Appels de fonction

- Un appel de fonction vient avec un surcoût (passage des paramètres par registres et ou la pile, instructions de saut)
- Est-ce que pour autant il ne faut pas faire d'appels de fonction ?
  - Parfois il vaut mieux éviter
  - Cela dépend d'où est placé l'appel de fonction

```
void stencil(const float *A, float *B, const int i, const int j, const int cols) {
    B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] +
                    A[(i  ) + (j  )*cols] +
                    A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];
}

void main() {
    for (int j = 1; j < rows -1; j++) // row
        for (int i = 1; i < cols -1; i++) // column
            stencil(A, B, i, j, cols); // ici on appelle la fonction très souvent
}
```

Inlining

- Inliner une fonction revient à remplacer l'appel de la fonction par le code de la fonction lui même
  - Comme cela on élimine le surcoût de l'appel à la fonction
- On peut le faire manuellement mais ce n'est pas une bonne idée...
- On peut le faire faire au compilateur, meilleur idée !
  - Utilisation du mot clef *inline* du langage C/C++
  - Option d'optimisation du compilateur (`-finline-function`)

Déroulage de boucle

- Le déroulage de boucle consiste à augmenter le pas de la boucle et à adapter le corps de la boucle en fonction
- Appelé *loop-unrolling* en anglais
- Une optimisation que peut parfois faire le compilateur (mais pas toujours)
- Plusieurs bénéfices
  - Diminue le temps passé dans le contrôle de boucle
  - Diminue le risque d'erreur de prédiction d'un branchement
  - Augmente les opportunités d'optimisation, expose potentiellement plus de parallélisme pour l'ILP, masquage de la latence des instructions
- Des défauts
  - Dégrade la lisibilité du code et augmente le risque de bugs (pas bon pour la maintenabilité)
  - Souvent il faut écrire un épilogue (code après la boucle)

Déroulage de boucle : exemple

Code initial sans déroulage de boucle

```
for (i = 0; i < n; i++) {
    D[i] = A[i] + B[i] + C[i];
}
```

Code avec déroulage de boucle d'ordre 2

```
for (i = 0; i < n; i += 2) {
    D[i+0] = A[i+0] + B[i+0] + C[i+0];
    D[i+1] = A[i+1] + B[i+1] + C[i+1];
}
```

Déroulage de boucle : exemple

3 Optimisations CPU mono-cœur

Code initial sans déroulage de boucle

```
for (i = 0; i < n; i++) {
    D[i] = A[i] + B[i] + C[i];
}
```

Code avec déroulage de boucle d'ordre 2

```
for (i = 0; i < n; i += 2) {
    D[i+0] = A[i+0] + B[i+0] + C[i+0];
    D[i+1] = A[i+1] + B[i+1] + C[i+1];
}
```

- Si on suppose que `n` vaut 3, alors le code avec déroulage est faux !

Déroulage de boucle : exemple

Code initial sans déroulage de boucle

```c
for (i = 0; i < n; i++) {
    D[i] = A[i] + B[i] + C[i];
}
```

Code avec déroulage de boucle d'ordre 2

```c
1 for (i = 0; i < n; i += 2) {
2     D[i+0] = A[i+0] + B[i+0] + C[ +0];
3     D[i+1] = A[i+1] + B[i+1] + C[i+1];
4 }
5 if (n % 2)
6     D[n-1] = A[n-1] + B[n-1] + C[n-1];
```

- Si on suppose que `n` vaut 3, alors le code avec déroulage est faux !
- Il faut ajouter un épilogue L5-6

Déroulage de boucle et mélange

Code avec déroulage de boucle d'ordre 2

```
for (i = 0; i < n; i += 2) {
    D[i+0] = A[i+0] + B[i+0] + C[i+0];
    D[i+1] = A[i+1] + B[i+1] + C[i+1];
}
```

Code avec déroulage de boucle d'ordre 2 et mélange :

```
for (i = 0; i < n; i += 2) {
    d0 = A[i+0] + B[i+0];
    d1 = A[i+1] + B[i+1];
    D[i+0] = d0 + C[i+0];
    D[i+1] = d1 + C[i+1];
}
```

- Aussi appelé *Unroll & Jam* en anglais
- Permet de casser des dépendances de données
- On peut commencer à calculer un bout de `D[i+1]` (dans la variable `d1`) alors que `D[i+0]`, n'a pas encore été complètement calculé
- C'est une optimisation que peut parfois faire le compilateur

Rotation de variables

3 Optimisations CPU mono-cœur

Somme de 3 points glissante :

```
for (i = 1; i < n -1; i++) {
    B[i] = A[i-1] + A[i+0] + A[i+1];
}
```

Somme de 3 points glissante avec déroulage à l'ordre 3 :

```
for (i = 1; i < n -1; i += 3) {
    B[i+0] = A[i-1] + A[i+0] + A[i+1];
    B[i+1] = A[i+0] + A[i+1] + A[i+2];
    B[i+2] = A[i+1] + A[i+2] + A[i+3];
}
// pas d'épilogue pour simplifier
```

Somme de 3 points glissante avec rotation de variables :

```
a0 = A[0];
a1 = A[1];
a2 = A[2];
a3 = A[3];
for (i = 1; i < n -1; i += 3) {
    // un seul accés en lecture à A
    a4 = A[i+4];
    B[i+0] = a0 + a1 + a2;
    B[i+1] = a1 + a2 + a3;
    B[i+2] = a2 + a3 + a4;
    // rotation sur les variables b0 et b1
    a0 = a1;
    a1 = a2
    a2 = a3;
    a3 = a4;
}
```

Fusion de boucles

3 Optimisations CPU mono-cœur

Deux boucles indépendantes :

```
for (i = 0; i < n; i++)
    D[i] = A[i] + B[i];
for (int i = 0; i < n; i++)
    E[i] = A[i] * C[i];
```

Fusion des deux boucles en une seule :

```
for (i = 0; i < n; i++) {
    D[i] = A[i] + B[i];
    E[i] = A[i] * C[i];
}
```

- Cela permet d'améliorer la réutilisation des données
- Dans l'exemple, la deuxième lecture de `A[i]` L2 sera forcément dans les caches
- Localité temporelle

Séparation en plusieurs boucles

- Découpage d'une boucle en plusieurs boucles
- L'opération inverse de la fusion de boucles
- Pour des raisons particulières (exemple : multi-threading)
- Simplifie ou élimine une dépendance en coupant la boucle en deux parties
- Parfois la pression sur les registres fait qu'il est plus intéressant d'avoir des boucles séparées

Instructions de branchement conditionnel

- Les instructions de branchement conditionnel (alias `if`, `switch`, etc) crées des bulles dans le pipeline du processeur
- Le pipeline ne peut pas opérer à pleine efficacité
- Il faut donc, autant que possible, éviter ces instructions dans les parties chaudes de notre code

Instructions de branchement conditionnel

Exemple d'un mauvais code à gauche et d'un code meilleur à droite :

```c
1  for (int i = 0; i < n; i++) {
2      if (i >= 1 && i < n -1) {
3          switch (i % 4) {
4              case 0: B[i] = A[i] * 0.3333f;
5              case 1: B[i] = A[i] + 1.3333f;
6              case 2: B[i] = A[i] - 0.7555f;
7              case 3: B[i] = A[i] * 1.1111f;
8              default: break;
9          }
10     }
11 }
```

```c
for (int i = 1; i < n -1; i += 4) {
    B[i+0] = A[i+0] + 1.3333f;
    B[i+1] = A[i+1] - 0.7555f;
    B[i+2] = A[i+2] * 1.1111f;
    B[i+3] = A[i+3] * 0.3333f;
}
```

- Le `if` L2 peut être supprimé en modifiant le début et la fin de la boucle
- Le `switch` L3 peut être évité en déroulant la boucle de 4

Accès mémoires

- Quand le code est limité par le débit mémoire, il faut faire très attention à la façon dont on accède aux données
- La bande passante mémoire est un facteur limitant très fort dans les architectures modernes
  - Il y a des mécanisme pour diminuer ce problème: utilisation d'instructions de *pre-fetching*
  - Les accès à la mémoire sont fait par ligne de mots (un mot = paquet de 32 bits)
  - Il est intéressant d'accéder à des données qui se suivent (localité spatiale)
  - Il faut réduire le nombre d'accès dans la RAM et favoriser les accès dans les caches (localité temporelle)

Exemple d'accès à la mémoire


```
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

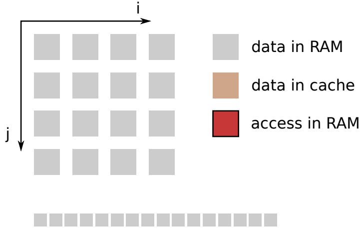


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire

3 Optimisations CPU mono-cœur

```
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

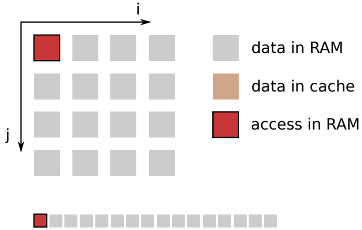


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire

3 Optimisations CPU mono-cœur

```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

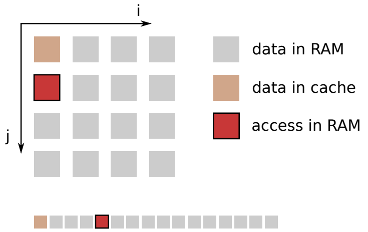


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire

3 Optimisations CPU mono-cœur

```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

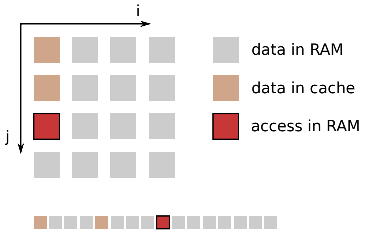


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d’accès à la mémoire


```
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

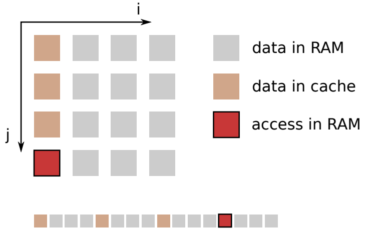


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire

3 Optimisations CPU mono-cœur

```
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

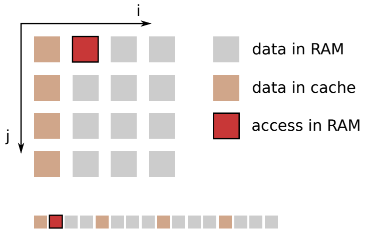


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire


```
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

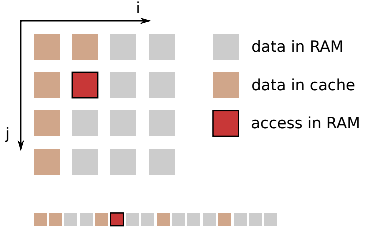


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire

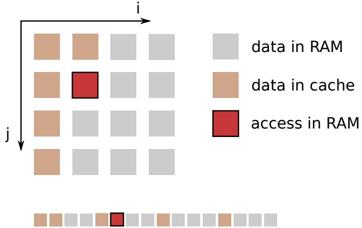


Figure: Vue logique et matérielle des accès à la mémoire

- Dans cette implémentation les accès aux données ne sont pas contiguës en mémoire
- Il y a un saut de 4 entre chaque accès (pas bon pour la localité spatiale)

Exemple d'accès à la mémoire : inversion


```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

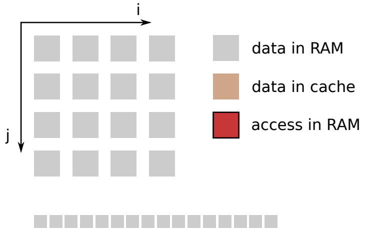


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion


```
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

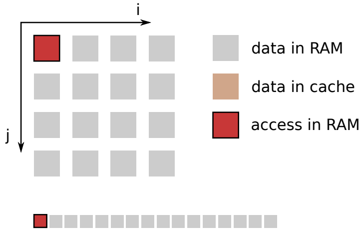


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion


```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

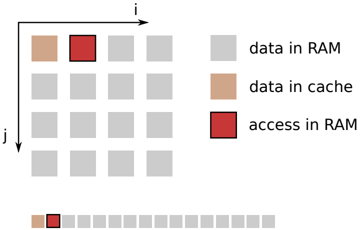


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion


```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

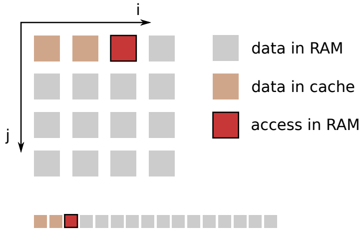


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion


```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

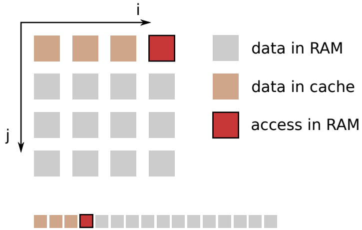


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion


```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

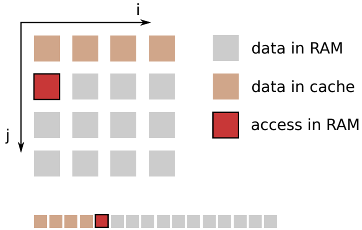


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion

3 Optimisations CPU mono-cœur

```c
for (int j = 0; j < n; j++) // row
    for (int i = 0; i < n; i++) // column
        C[i + j*n] = A[i + j*n] + B[i + j*n];
```

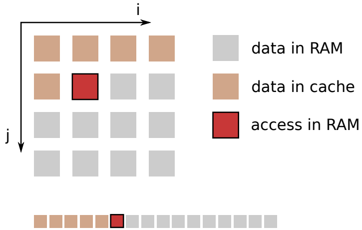


Figure: Vue logique et matérielle des accès à la mémoire

Exemple d'accès à la mémoire : inversion

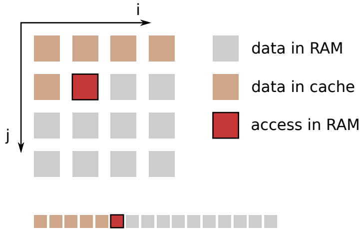


Figure: Vue logique et matérielle des accès à la mémoire

- Dans cette implémentation les accès sont contiguës en mémoire
  - Les lignes de cache sont complètement utilisées
  - Le débit mémoire est maximisé
- Les boucles en `i` et en `j` ont simplement été inter-changées

Cache blocking

- Dans beaucoup de cas, des données peuvent être réutilisées (localité spatiale)
- Prenons l'exemple d'un code *stencil* fonctionnant sur une grille 2D

```
for (int j = 1; j < rows -1; j++) // row
    for (int i = 1; i < cols -1; i++) // column
        B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] + // left, right
                        A[(i  ) + (j  )*cols] +                           // center
                        A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];  // top, bottom
```

Parcours SANS cache blocking


```c
for (int j = 1; j < rows -1; j++) // row
    for (int i = 1; i < cols -1; i++) // column
        B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] + // left, right
                        A[(i  ) + (j  )*cols] +                          // center
                        A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];  // top, bottom
```

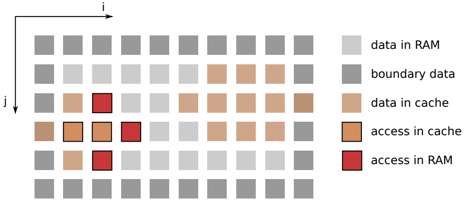


Figure: Vue logique de la grille 2D

Parcours SANS cache blocking

3 Optimisations CPU mono-cœur

```
for (int j = 1; j < rows -1; j++) // row
    for (int i = 1; i < cols -1; i++) // column
        B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] + // left, right
                        A[(i  ) + (j  )*cols] +                          // center
                        A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];   // top, bottom
```

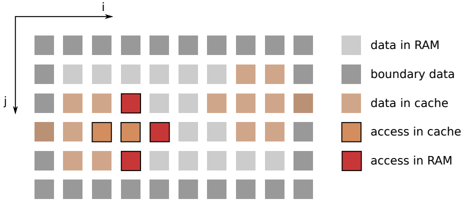


Figure: Vue logique de la grille 2D

Parcours SANS cache blocking


```
for (int j = 1; j < rows -1; j++) // row
    for (int i = 1; i < cols -1; i++) // column
        B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] + // left, right
                        A[(i  ) + (j  )*cols] +                         // center
                        A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];  // top, bottom
```

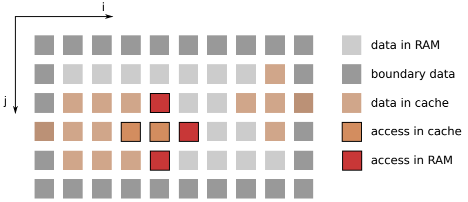


Figure: Vue logique de la grille 2D

Parcours SANS cache blocking


```
for (int j = 1; j < rows -1; j++) // row
    for (int i = 1; i < cols -1; i++) // column
        B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] + // left, right
                        A[(i  ) + (j  )*cols] +                          // center
                        A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];  // top, bottom
```

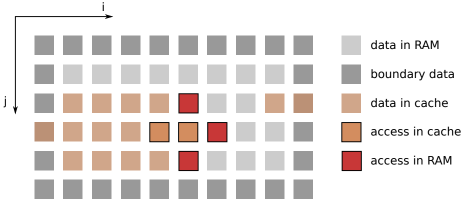


Figure: Vue logique de la grille 2D

Parcours SANS cache blocking

3 Optimisations CPU mono-cœur

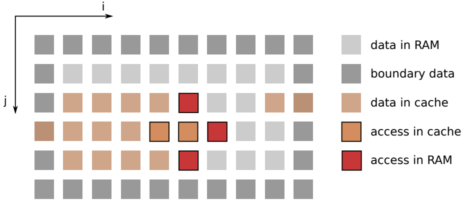


Figure: Logical 2D grid memory view

- À chaque incrément de `i` il y a 3 nouveaux accès dans la RAM et 2 accès dans le cache
- Peut-on diminuer le nombre d'accès à la RAM ?
  - Oui, en utilisant un parcours dit avec *cache blocking*
  - L'idée est de modifier le parcours des données pour maximiser la réutilisation

Parcours AVEC cache blocking

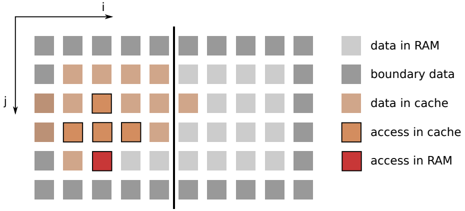


Figure: Vue logique de la grille 2D

Parcours AVEC cache blocking

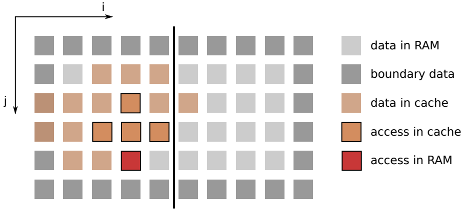


Figure: Vue logique de la grille 2D

Parcours AVEC cache blocking

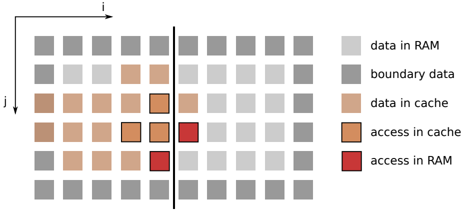


Figure: Vue logique de la grille 2D

Parcours AVEC cache blocking

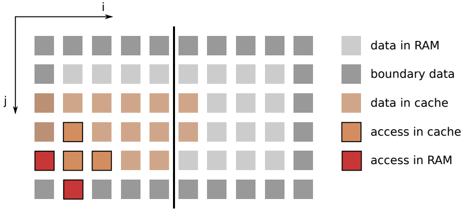


Figure: Vue logique de la grille 2D

Parcours AVEC cache blocking

3 Optimisations CPU mono-cœur

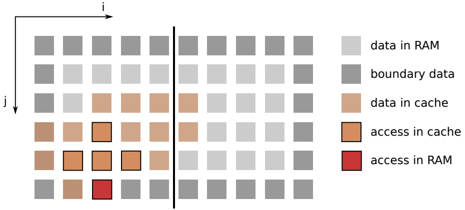


Figure: Vue logique de la grille 2D

Parcours AVEC cache blocking

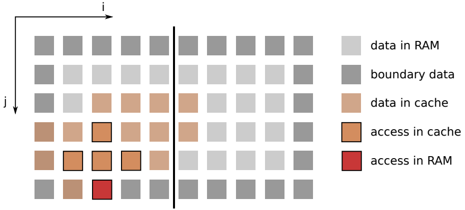


Figure: Logical 2D grid memory view

- Avec le cache blocking le nombre d'accès à la RAM est réduit
  - Il reste en moyenne un seul accès à la RAM!
  - Nous avons découpé la grille en plusieurs blocs (ici 2), séparation verticale

Cache blocking : définir la taille des blocs

- La taille d'un bloc dépend du problème
- Pour le code stencil précédent, la taille d'un bloc peut être définie comme suit :

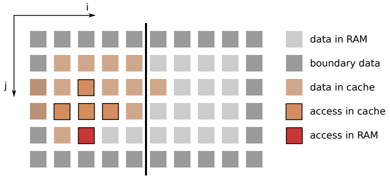
$$blockSize = \frac{sizeOfCache}{2 \times 3 \times sizeOfData},$$

avec $sizeOfCache$ la taille du plus grand cache (L3) en octets et $sizeOfData$ la taille des données (simple précision = 4 octets, double précision = 8 octets).

- On divise par 2 parce que les caches fonctionnent de manière optimale quand on utilise la moitié (recette de grand mère),
- On divise par 3 parce que l'on doit garder 3 lignes en cache dans le stencil présenté
- Notez que si $blockSize \geq cols$ alors le cache blocking ne sert à rien

Cache blocking : implémentation

3 Optimisations CPU mono-cœur




```c
#define SIZE_OF_CACHE_L3 96 // on suppose un cache L3 de 96 octets pour l'exemple
int blockSize = SIZE_OF_CACHE_L3 / (2 * 3 * sizeof(float)); // (96 / 24) = 4

for (int iOff = 1; iOff < cols -1; iOff += blockSize) { // boucle sur les blocs verticaux
    blockSize = min(cols -1 - iOff, blockSize); // réduction de la taille du bloc si besoin
    for (int j = 1; j < rows -1; j++) // row
        for (int i = iOff; i < iOff + blockSize; i++) // column
            B[i + j*cols] = A[(i-1) + (j  )*cols] + A[(i+1) + (j  )*cols] +
                            A[(i  ) + (j  )*cols] +
                            A[(i  ) + (j-1)*cols] + A[(i  ) + (j+1)*cols];
}
```

Q&R

Merci pour votre écoute !
Avez-vous des questions ?