

## Systèmes Distribués

# Systèmes parallèles & architecture

**INFO – AGROTIC**

Last update: oct. 2025



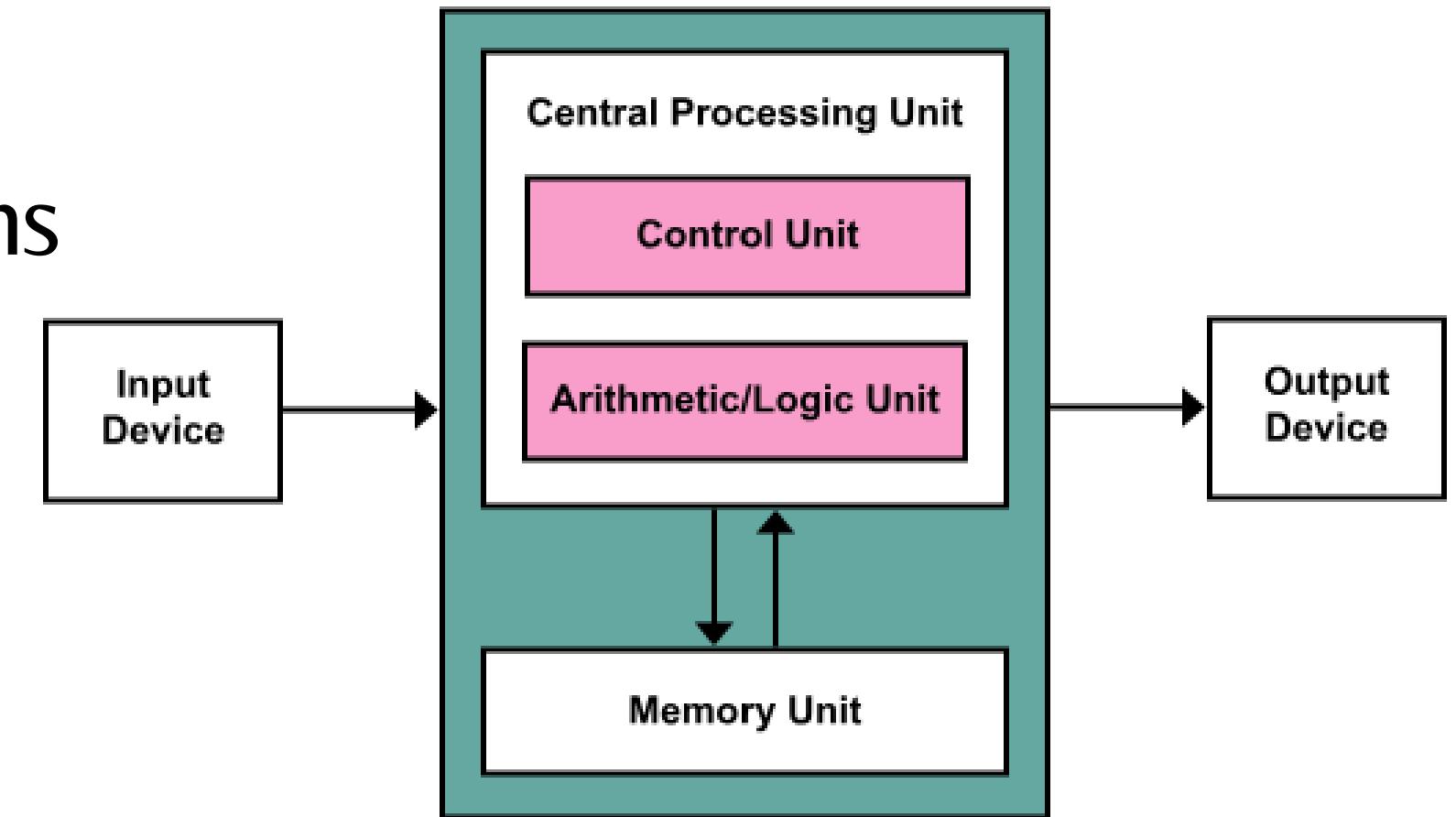
# **De Von Neumann aux Multi-coeurs**

**ARCHITECTURE**

# Von Neumann- Architecture simplifiée

## Principe fondamental

- Programme et données sont stockés dans la même mémoire.
- Le CPU lit séquentiellement les instructions à exécuter.

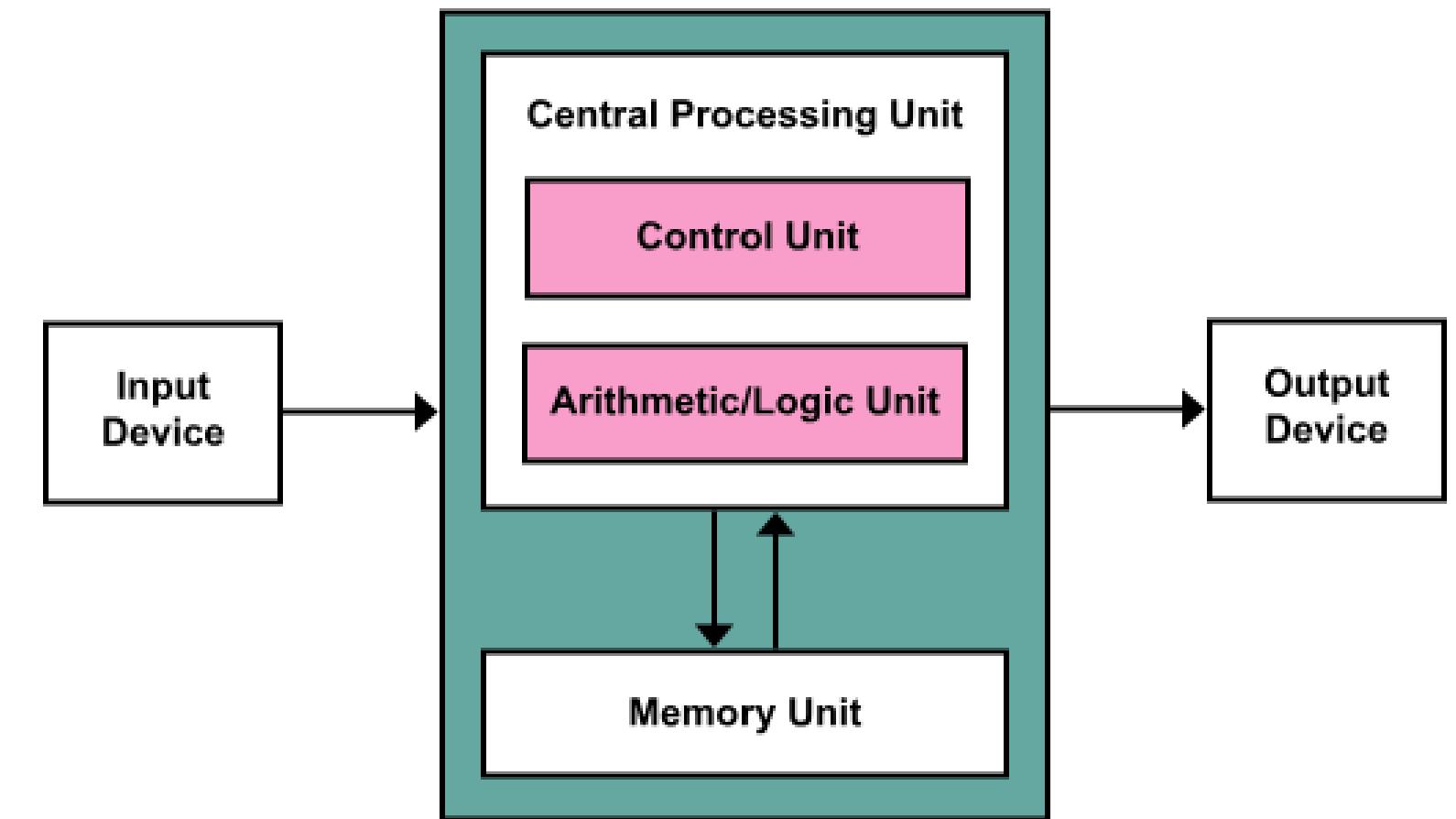


Architecture Von Neumann, 1945

# Von Neumann- Principale limite

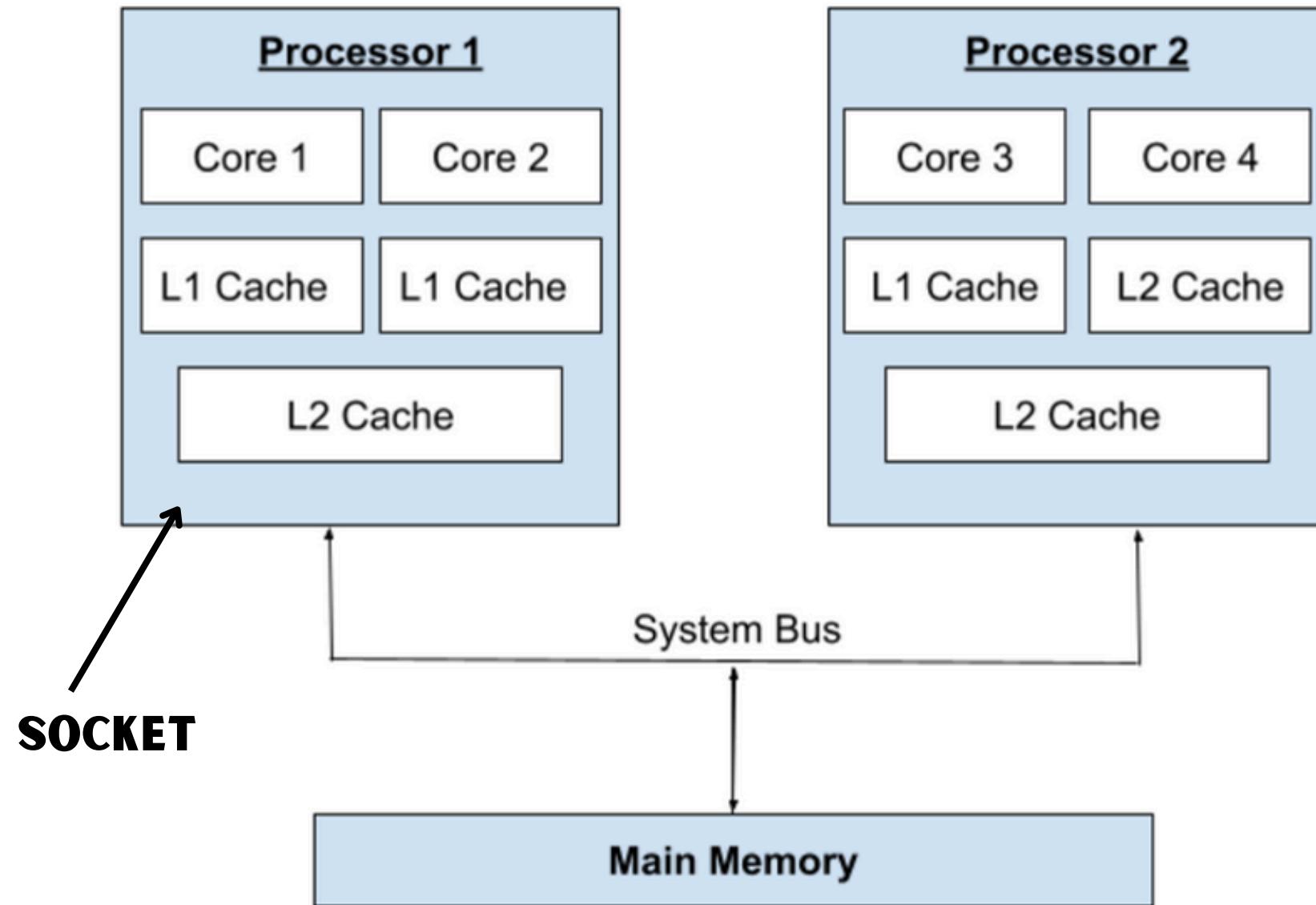
## Goulot d'étranglement

Le CPU ne peut accéder qu'à une seule donnée ou instruction à la fois → ralentissements possibles.



Architecture Von Neumann, 1945

# Multi-cœurs- Architecture simplifiée



## Principe fondamental

- Chaque cœur est l'équivalent d'un CPU disposant de registres dédiés comme sur Von Neumann.
- Chaque cœur a son propre cache L1.
- Le cache L1 est généralement partagé en deux parties: L1 pour données, L1 pour Instructions.
- Les cœurs dans un socket se partagent le cache L2.
- Tous les cœurs ont accès à la mémoire principale.

# Du processeur moderne

Ce qu'il faut savoir !!

De Von Neuman à nos jours !! Du changement

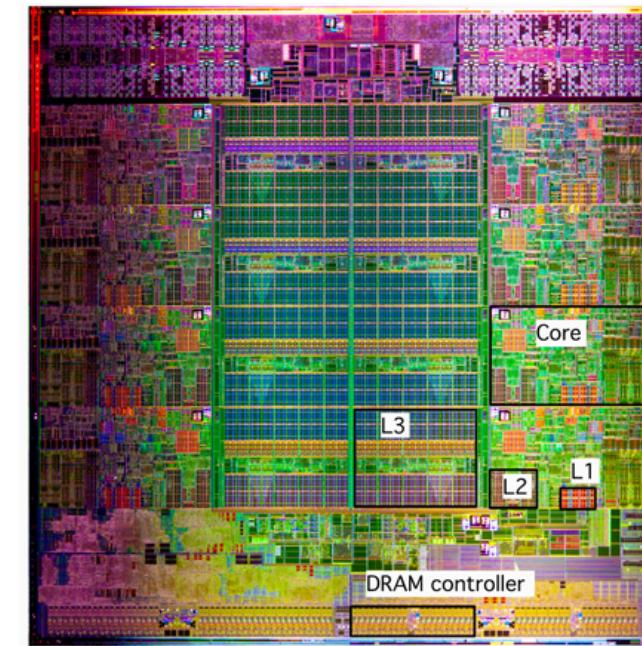


Figure 1.1: The Intel Sandy Bridge processor die.

- Plusieurs FPUs : Le processeur moderne est équipé de plusieurs unités de calcul flottants (FPU) fonctionnant en parallèle.
- Out-of-order instructions : les processeurs modernes, contrairement à Von Neumann, peuvent exécuter les instructions dans un ordre autre que celui défini par l'utilisateur.
- Si le processeur identifie des opérations d'addition et de multiplication indépendantes, il peut les exécuter parallèlement, doublant ainsi les performances du processeur (Pipelining).

Addition et multiplication sont bien optimisées par le CPU moderne (résultat produit en chaque cycle) alors que la division reste toujours non optimisée (environ une vingtaine de cycles)

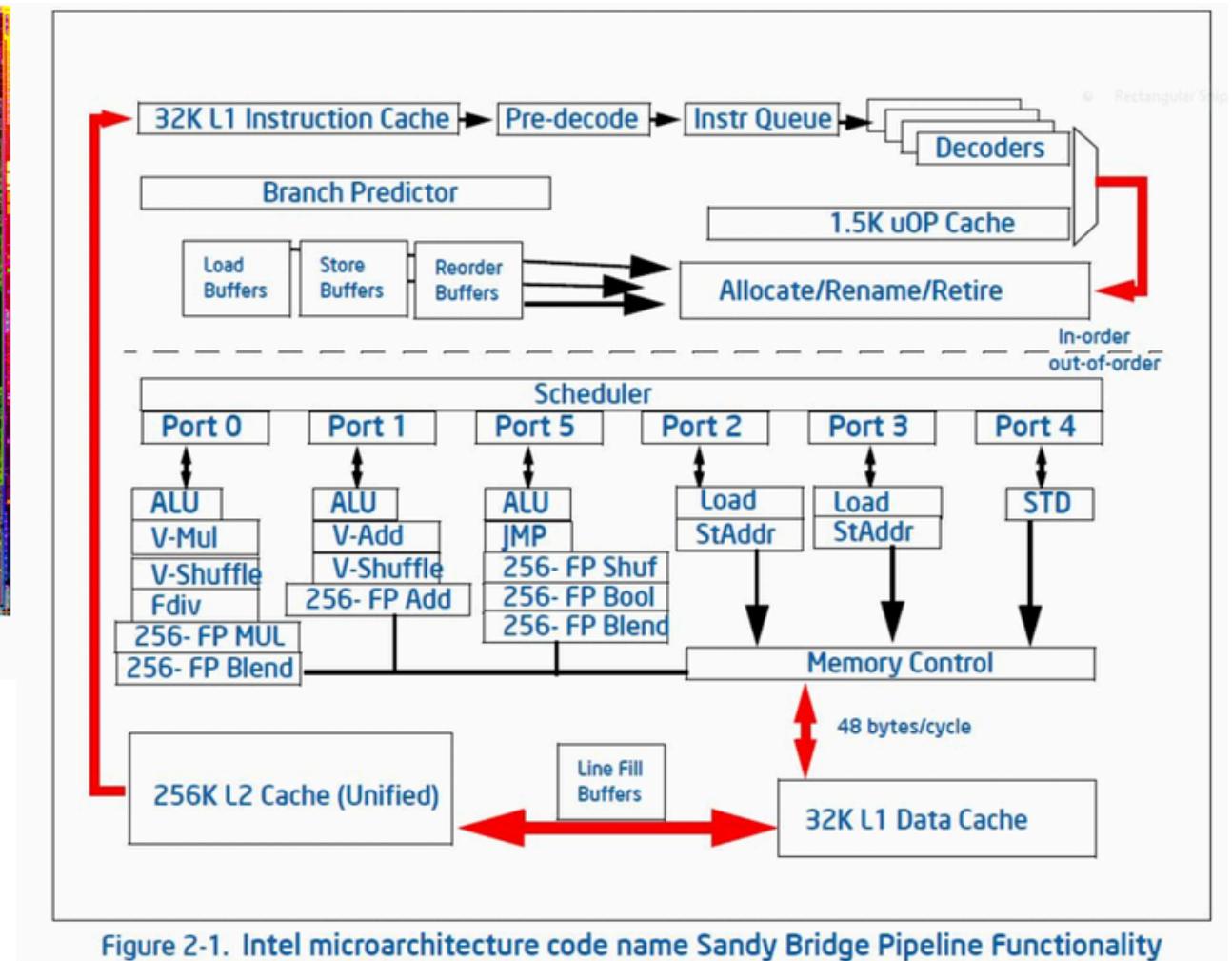
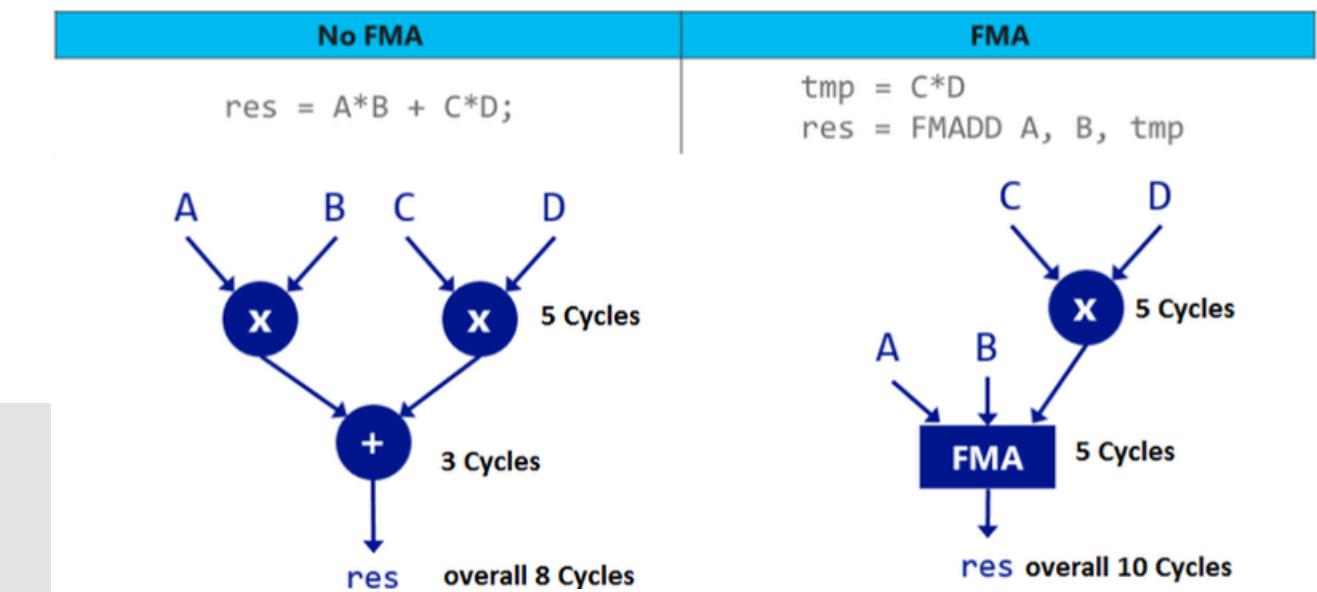


Figure 2.1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

Fused Multiply-Add (FMA)



# ALU: Unité Arithmétique et Logique

**ALU – Arithmetic and Logic Unit**

L’ALU est l’unité du processeur chargée de réaliser toutes les opérations arithmétiques et logiques sur les entiers.



**Remarques :**

- L’ALU travaille sur des entiers (contrairement à la FPU pour les flottants).
- Elle est au cœur de l’exécution des instructions machine.
- Elle peut être pipelinisée pour exécuter plusieurs opérations en parallèle.

# ALU: Unité Arithmétique et Logique

## Composants principaux :

- AU (Arithmetic Unit) qui effectue les calculs → addition, soustraction, multiplication, division.
- LU (Logic Unit) qui réalise les opérations logiques → AND, OR, NOT, XOR, etc.
- Adders/Subtractors : circuits de base pour addition et soustraction.
- Compareurs & Shift Units : pour comparaisons et décalages binaires.

# **ALU: Unité Arithmétique et Logique**

## **Fonctions principales :**

- Opérations arithmétiques (add, sub, mul, div)
- Opérations logiques (AND, OR, NOT, XOR...)
- Comparaisons ( $>$ ,  $<$ ,  $=$ )
- Décalages de bits (left/right shift)

# **ALU: Unité Arithmétique et Logique**

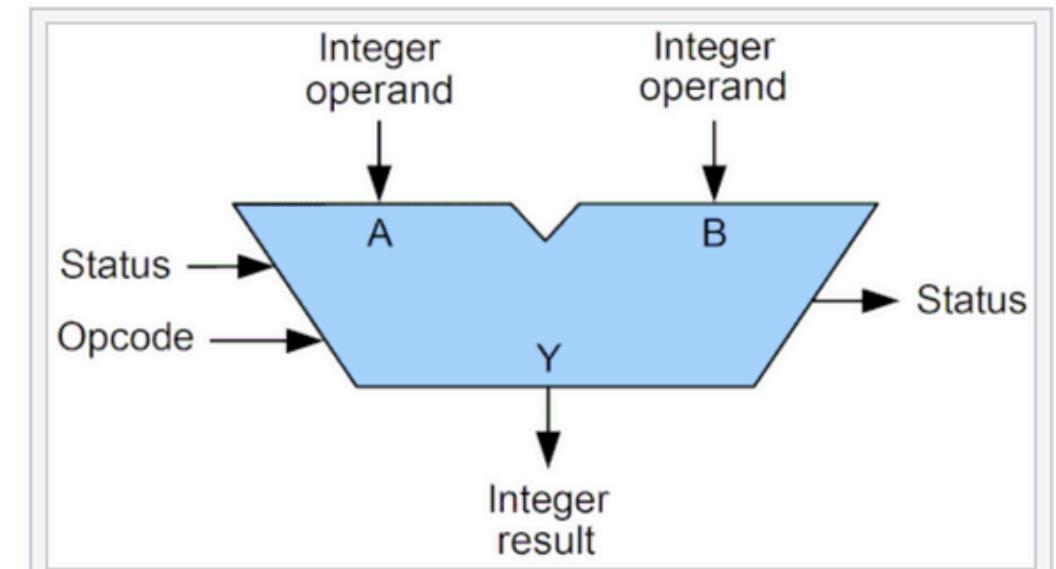
## **Fonctionnement :**

1. Reçoit les données et signaux de contrôle.
2. Sélectionne le circuit approprié selon l'opération demandée.
3. Exécute l'opération et envoie le résultat à la mémoire ou à la sortie.

# ALU: Unité Arithmétique et Logique

## Importance :

- Moteur de traitement du CPU
- Haute précision et rapidité
- Permet la prise de décision logique et les calculs complexes



Représentation symbolique d'une UAL et de ses signaux d'entrée et de sortie, indiqués par des flèches pointant respectivement vers l'intérieur et vers l'extérieur de l'UAL. Chaque flèche représente un ou plusieurs signaux. Les signaux de contrôle entrent par la gauche et les signaux d'état sortent par la droite ; les données circulent de haut en bas.

Wikipédia

# FPU: Unité de calcul flottant



## FPU – Floating Point Unit

La FPU est une unité matérielle du processeur spécialisée dans le traitement des nombres à virgule flottante (ex : 3.14, -2.718, etc.).

## Rôle principal

- Effectuer des opérations arithmétiques complexes :
  - ➤ Addition, soustraction, multiplication, division
  - ➤ Racine carrée, trigonométrie, etc.
- Manipuler des nombres flottants (simple (32bits), double (64bits), ou quadruple précision (128bits))

# FMA vs Non-FMA

La distinction FMA vs Non-FMA est essentielle en calcul numérique et en architecture processeur.

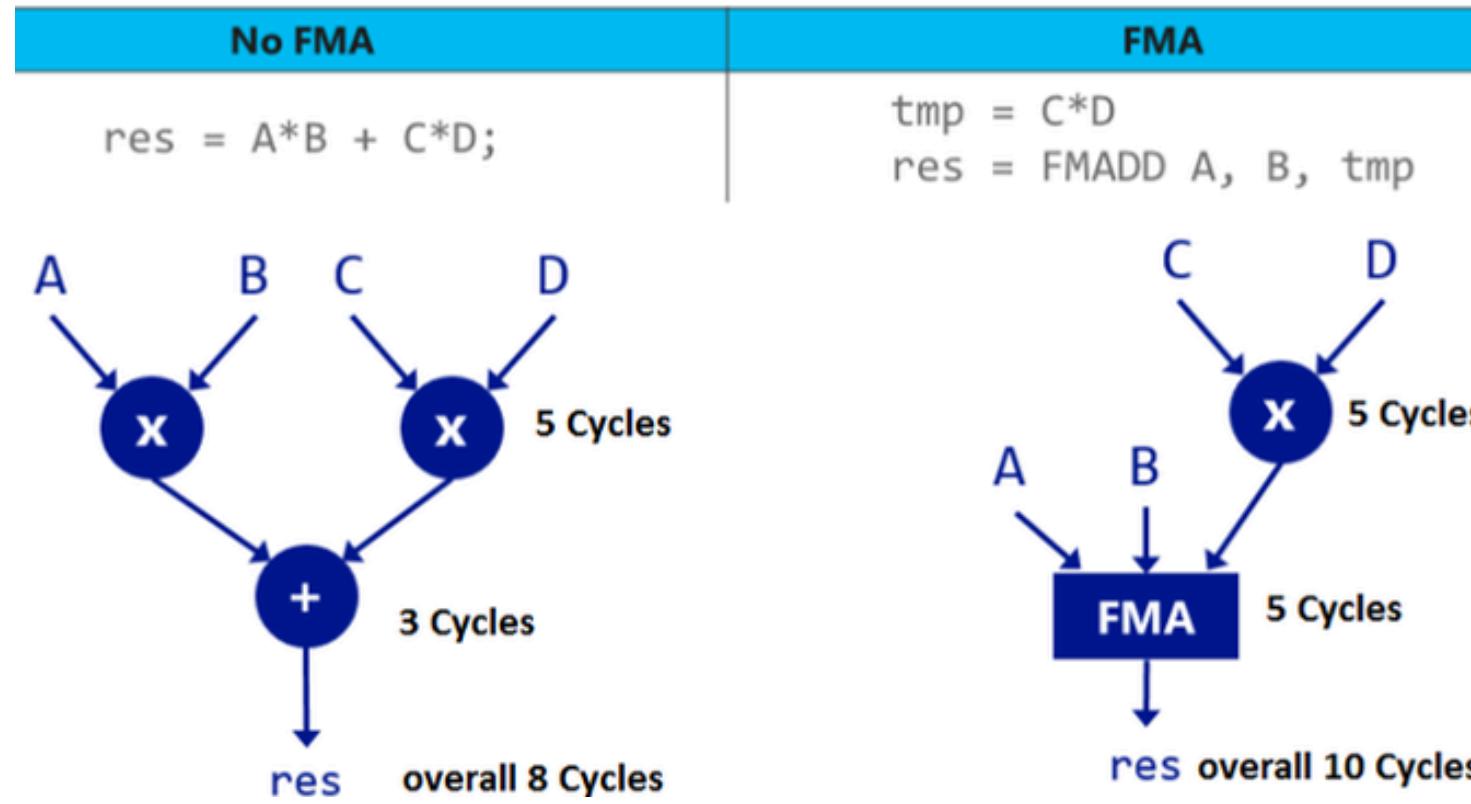
Voici une explication claire :

- ◆ FMA (Fused Multiply-Add): FMA est une instruction matérielle qui exécute en une seule étape

$$\text{FMA}(a, b, c) = (a \times b) + c$$

- ◆ Non-FMA: Effectuer séparément la multiplication puis l'addition

$$(a \times b) \quad \text{puis} \quad + c$$



main.c

```
1 #include <stdio.h>
2 #include <math.h> // pour fma()
3
4 int main() {
5     double a = 1e20;
6     double b = 1.000000001;
7     double c = -1e20;
8
9     // Non-FMA : (a * b) + c
10    double result_non_fma = (a * b) + c;
11
12    // FMA : fma(a, b, c) = (a * b) + c avec une seule opération, sans arrondi intermédiaire
13    double result_fma = fma(a, b, c);
14
15    // Affichage
16    printf("Résultat NON-FMA : %.10f\n", result_non_fma);
17    printf("Résultat FMA : %.10f\n", result_fma);
18
19    return 0;
20}
21
```

Résultat NON-FMA : 10000007168.0000000000  
Résultat FMA : 1000000827.4037094116

...Program finished with exit code 0  
Press ENTER to exit console.

# Boucle d'exécution d'un CPU

TANT QUE (le programme n'est pas terminé)

1. PC → Adresse mémoire instruction

// Le registre compteur ordinal (PC) donne l'adresse de l'instruction à charger

2. Instruction ← Mémoire (via Cache L1 → Cache L2 → RAM)

// L'unité de contrôle récupère l'instruction

3. Décoder l'instruction

// L'unité de contrôle identifie l'opération à effectuer

4. SI instruction nécessite des données

- a. Lire les opérandes

- Chercher dans les “registres”

- Sinon, chercher dans le “cache L1”, puis “L2”, puis dans la “RAM”

- Charger les données dans les “registres”

5. Exécuter l'instruction

- Par l'unité arithmétique et logique (ALU) ou “FPU” selon le type

6. Écrire le résultat

- Vers les “registres” si besoin

- Ou vers “cache → mémoire” si c'est une écriture mémoire

7. Incrémenter PC

**Processseurs 8 bits, 16 bits, 32 bits,  
64 bits – Que signifient-ils ?**

# 8 bits, 16 bits, 32 bits, 64 bits – Que signifient-ils ?

Les processeurs sont souvent classés selon la taille des données qu'ils peuvent traiter en une seule unité. Cela influence plusieurs aspects :

- Largeur du bus mémoire : un processeur 64 bits peut, par exemple, charger un nombre flottant 64 bits (double précision) en un seul cycle, alors qu'un processeur plus ancien pourrait devoir le charger en plusieurs parties.
- Adressage mémoire : si les adresses sont limitées à 16 bits, on ne peut accéder qu'à 64 000 octets (64 Ko). Les premiers PC utilisaient des systèmes de segmentation pour contourner cette limite, combinant un numéro de segment et un déplacement dans ce segment.

# 8 bits, 16 bits, 32 bits, 64 bits – Que signifient-ils ?

- Taille des registres : (notamment les registres entiers, utilisés pour manipuler les adresses) Plus les registres sont larges, plus on peut traiter de données en une seule instruction. Les **registres flottants** sont souvent plus grands, comme les registres 80 bits dans l'architecture x86.
- Taille des nombres flottants : une unité arithmétique conçue pour des **nombres** de 64 bits (double précision) peut parfois traiter des nombres de 32 bits (simple précision) encore plus rapidement.

# Evolution des processeurs Intel de 4004 à Pentium

C'ÉTAIT AVANT LES COEURS !!

processeur	Date	nombre de transistors	fréquence quartz	fréquence max (MHz)	bits	taille de la gravure (microns)
4004	nov-71	2300	0,108	0,108	4	pas de donnée
4040	févr-72	2300	0,747	0,747	4	pas de donnée
8008	avr-72	3500	0,3	0,3	8	pas de donnée
8080	avr-74	6000	2	2	8	pas de donnée
8086	juin-78	29000	5	10	16	pas de donnée
80286	févr-82	134000	6	12,5	16	pas de donnée
80386 DX	oct-85	275000	16	33	32	pas de donnée
80486 DX	avr-89	1200000	25	50	32	pas de donnée
Pentium P5	mars-93	3100000	60	66	64	1
Pentium	mars-93	3300000	90	120	32	0,6
Pentium pro	oct-95	5500000	150	200	32	0,6
Pentium II	juil-97	7500000	200	450	64	0,35
Pentium III	mars-99	29000000	450	1000	128	0,18
Pentium IV	nov-00	42000000	1400	1500	128	0,13

# Quatre grandes ères d'évolution des CPUs

- 1971–1985 : débuts du microprocesseur (4 bits → 32 bits)
- 1985–2000 : montée en fréquence et complexité (pipeline, cache, FPU)
- 2000–2010 : transition vers le multicœur
- 2010–2025 : Architectures hybrides et basse consommation

# Pipelining

# ILP (Instruction Level Parallelism)

Les CPUs modernes sont capables d'identifier les instructions indépendantes d'un programme et les faire exécuter parallèlement.

## Généralisation du pipeline

Gérer plusieurs opérations à des étapes différentes d'exécution.

## Prédiction de boucle

“Deviner” qu'une condition sera satisfaite et exécuter l'action anticipée

## Out-of-order

Réarranger les instructions indépendantes pour mieux gagner en efficacité

## Prefetching

Anticiper sur la lecture des données futures

Remarque : Les pipeline ne sont plus limités aux FPUs mais intègrent tout type d'instruction

# Pipelining

- Les FPU sur les processeurs modernes incorporent un pipeline
- Des opérations indépendantes sont nécessaires pour tirer partie d'un pipeline
- En calcul scientifique, IA, graphisme, les FPU doivent exécuter des millions d'opérations flottantes par seconde. Le pipeline FPU permet d'avoir un débit maximal, même si chaque opération prend plusieurs cycles.

Cycle:      1      2      3      4      5

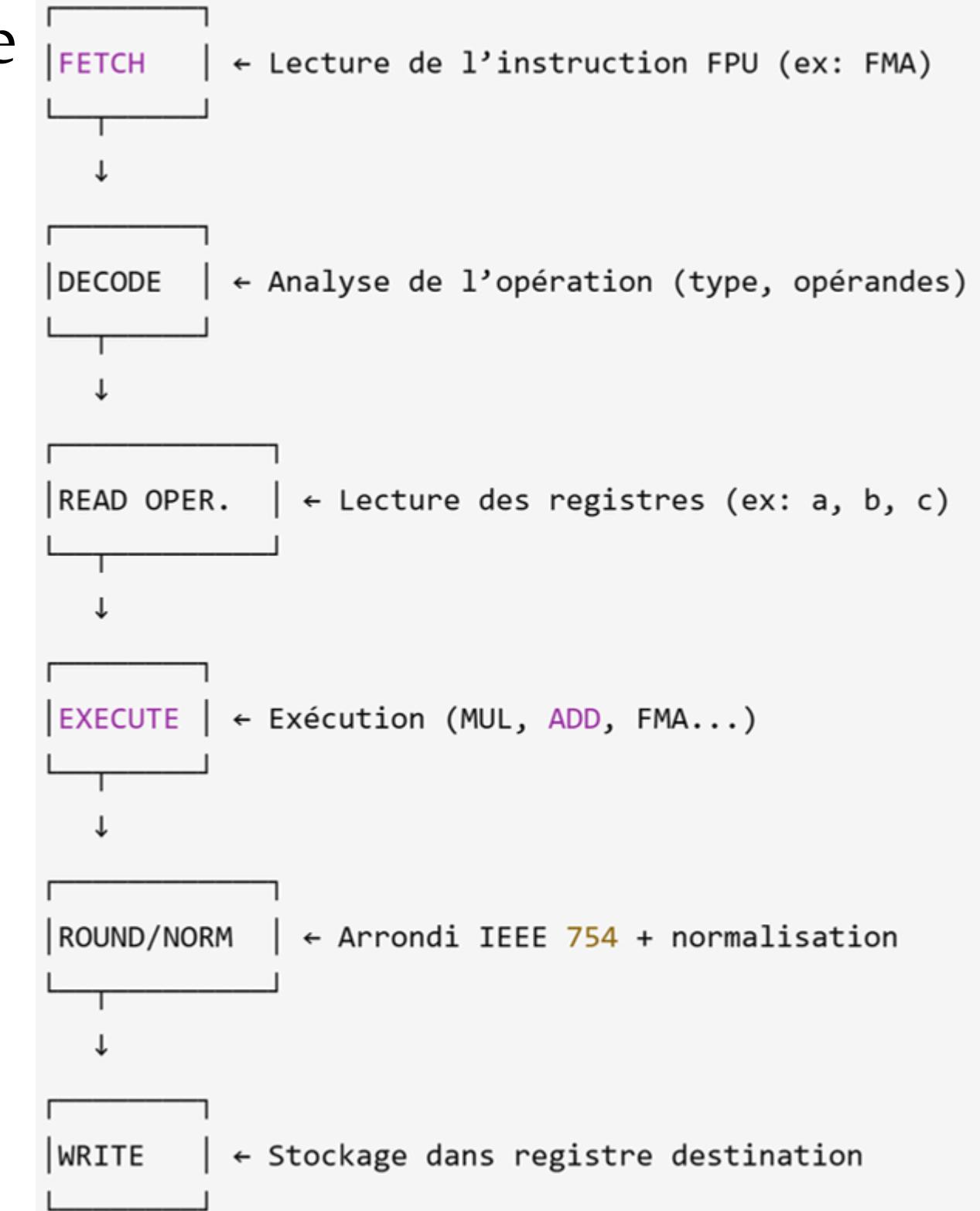
Opération:

FMA1      Fetch → Decode → Read → Execute → Write

FMA2      Fetch → Decode → Read → Execute → Write

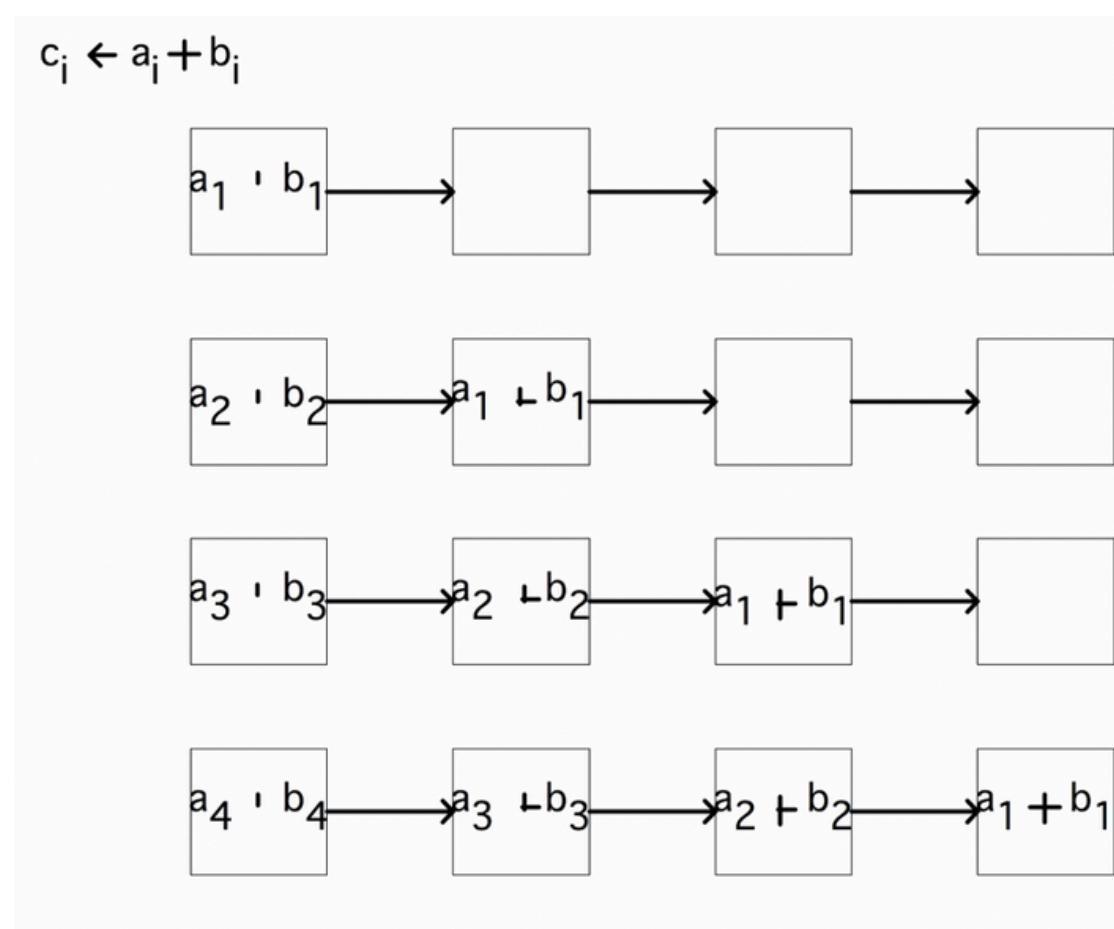
FMA3      Fetch → Decode → Read → Execute → Write

## Exemple: Pipeline en 6 étapes



# Pipelining

- Sur un FPU sans pipeline, le temps pour produire un résultat est exprimé par :  $t(n) = n\ell\tau$ 
    - $\ell$  est le nombre de stages, et  $\tau$  the clock cycle time
  - Sur un FPU avec pipeline, le temps pour produire un résultat est donné par :  $t(n) = [s + \ell + n - 1]\tau$ 
    - $s$  est un facteur de coût.



# Illustration schématique d'un calcul en pipeline

S'assurer de l'indépendance des instructions est cruciale afin de tirer partie des capacités que présentent les CPUs vecteurs :

$$\forall_i : a_i \leftarrow b_i + c_i$$

## Addition indépendante

$$\forall_i : a_i \leftarrow a_i b_i + c_i$$

# Addition dépendante de l'inst. précédent

# Pipelining

```
for (i) {  
    x[i+1] = a[i]*x[i] + b[i];  
}
```

**Question** : Pourquoi cette instruction ne peut pas être exploité par un pipeline ?

**Réponse** : le résultat de l'instruction suivant dépend du input du résultat précédent.

## Solution

Doublement récursif : dériver une expression qui calcule  $x[i+2]$  à partir de  $x[i]$  sans faire intervenir  $x[i+1]$ .

- effectuer quelques calculs préliminaires
- calculer  $x[i], x[i+2], x[i+4], \dots$ , et à partir de ceux-ci,
- calculer les termes manquants  $x[i+1], x[i+3], \dots$

# Pipelining

```
for (i) {  
    x[i+1] = a[i]*x[i] + b[i];  
}
```

**Question** : Pourquoi cette instruction ne peut pas être exploité par un pipeline ?

**Réponse** : le résultat de l'instruction suivant dépend du input du résultat précédent.

## Solution

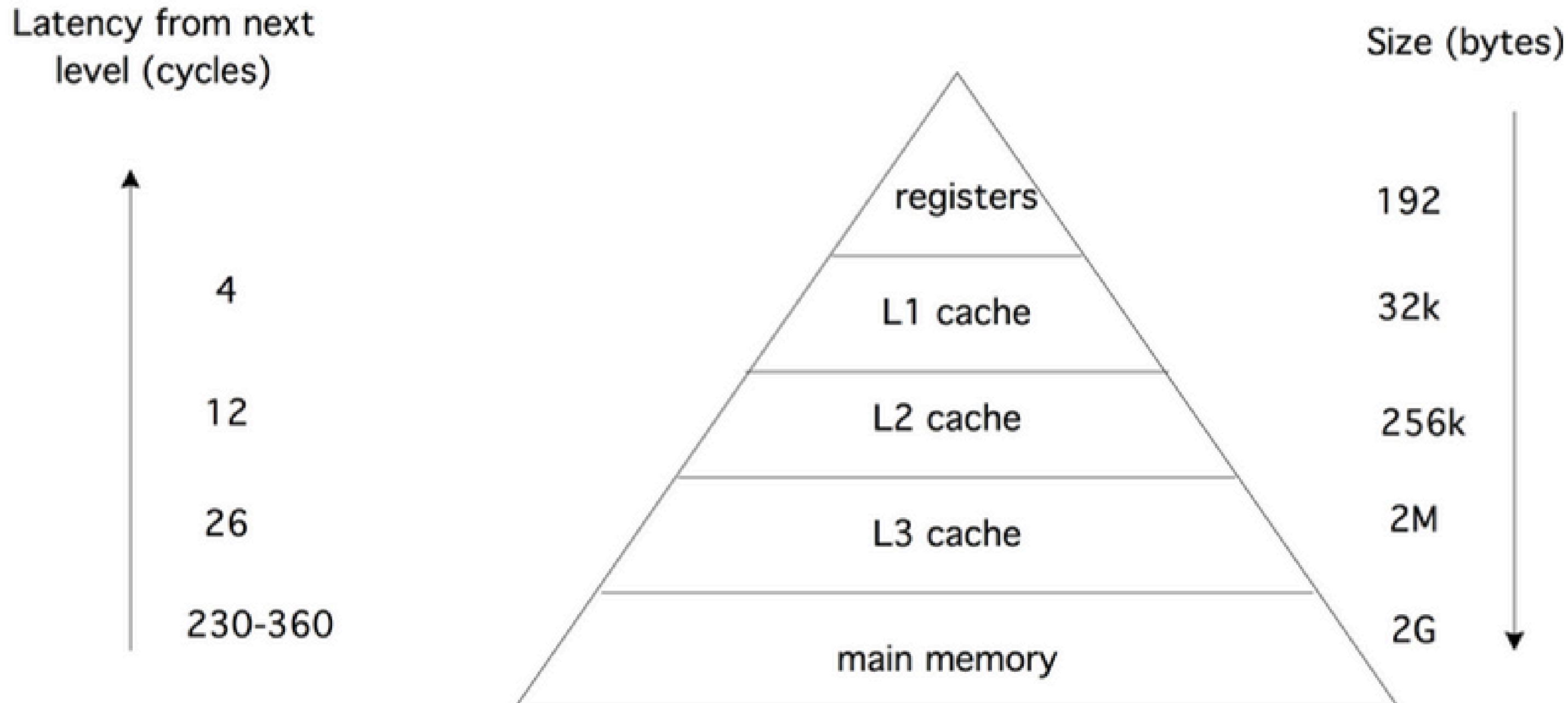
### Loop Unrolling

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

# Architecture mémoire

CPU MODERNES

# Hiérarchie des mémoires, vitesse et taille



*Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size.*

# Mémoires cache

## GÉNÉRALITÉS

- L1, L2, L3 (et parfois L4) selon les processeurs.
- L1 cache : très rapide, petit ( $\sim 16$  Ko), privé à chaque cœur.
- L2 cache : plus grand (jusqu'à quelques Mo), plus lent, parfois partagé.
- L3/L4 : encore plus grands, souvent partagés entre cœurs, hors du cœur CPU.

# Mémoire principale - RAM

## GÉNÉRALITÉS

- Mémoire RAM (mémoire principale) :
  - plus lente que les caches, mais beaucoup plus grande (Go à To).
  - Accessible par tous les cœurs.
  - Stocke les données et instructions en cours d'utilisation.
  - Temps d'accès entre +10 à +100 nanosecondes.

# Disque dur

## GÉNÉRALITÉS

- Disque dur / SSD (stockage secondaire) :
  - Beaucoup plus lent que la RAM (millisecondes pour les DD, microsecondes pour les SSD).
  - Très grande capacité (plusieurs To).
  - Sert à stocker programmes et données de façon permanente.
  - Les données doivent être chargées en RAM pour être utilisées par les cœurs de CPU.

# Rôle des caches

- **Caches – Mémoires intermédiaires entre registres et RAM**
  - Situés entre les registres et la mémoire principale.
  - Fournissent un accès plus rapide (faible latency, haut débit).
  - Conservent temporairement les données récemment utilisées.
- **Pourquoi utiliser des caches ?**
  - Si une donnée est réutilisée peu de temps après son premier accès, elle est probablement encore en cache.
  - Elle peut alors être rechargée rapidement, sans passer par la mémoire lente.

# Rôle des caches : exemple

Exemple simplifié

```
Instruction 1 : charger x → registre ; utiliser x
```

```
...
```

```
Instruction 2 : recharger x ; utiliser x
```

→ Sans cache : chaque accès recharge x depuis la RAM (lenteur)

→ Avec cache : le 2<sup>e</sup> accès recharge x directement depuis le cache → beaucoup plus rapide

# Les caches et le *hardware* !

## Remarques importantes

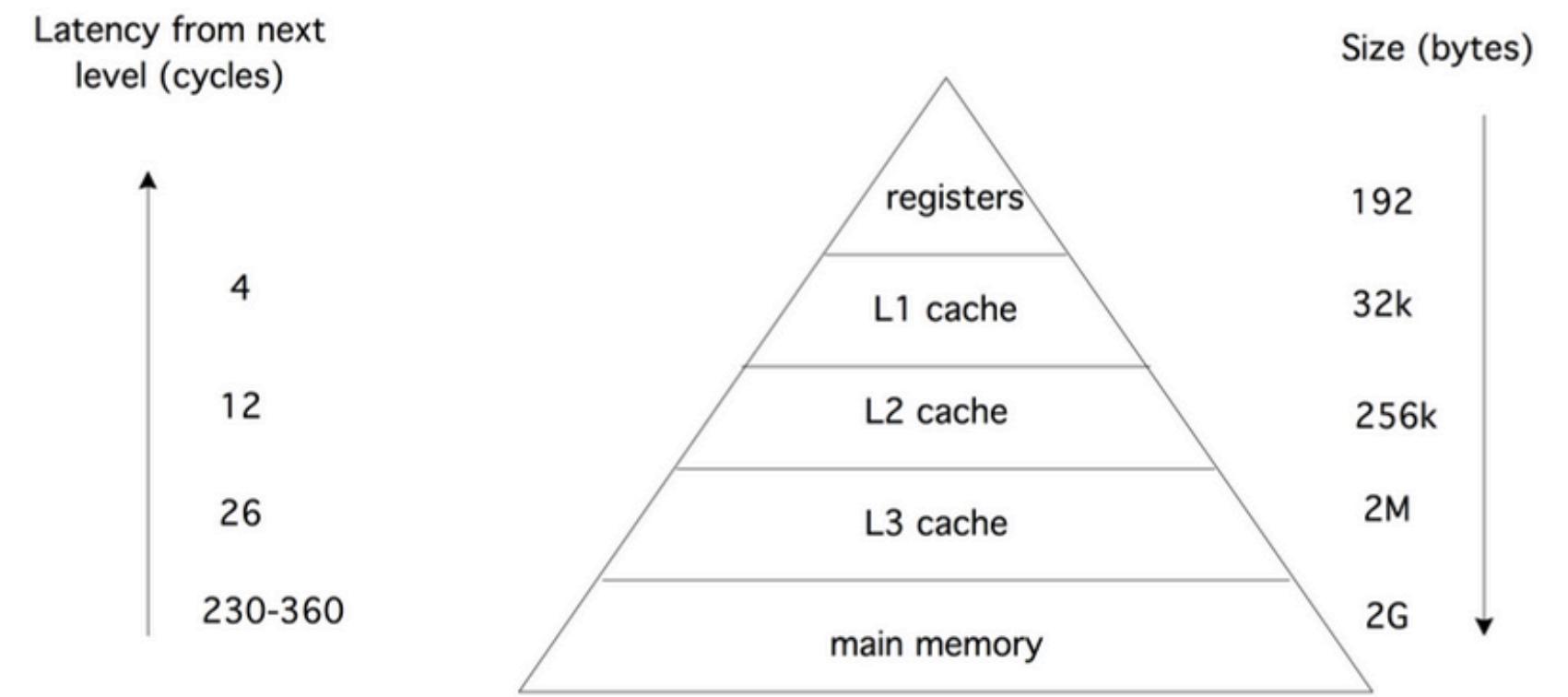
- Les caches sont gérés automatiquement par le matériel: l'utilisation et la réutilisation des caches se fait sans l'intervention du programmeur. Cependant le programmeur pourrait influencer l'utilisation de caches.
- Leur petite taille limite le temps pendant lequel une donnée y reste.

# Niveaux de cache, vitesse et taille

## Hiérarchie des caches

L1, L2, L3 (et parfois L4) selon les processeurs.

- L1 cache : très rapide, petit ( $\sim 16$  Ko), privé à chaque cœur.
- L2 cache : plus grand (jusqu'à quelques Mo), plus lent, parfois partagé.
- L3/L4 : encore plus grands, souvent partagés entre coeurs, hors du cœur CPU.



*Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size.*

# Algorithme simple d'accès mémoire (hiérarchisé)

Supposons que le processeur a besoin d'une donnée pour effectuer un calcul :

**Le processeur cherche d'abord dans le cache L1**

**Si donnée "non retrouvée" alors**

**Chercher dans le cache L2**

**Sinon**

**Chercher dans L3 (s'il existe)**

**Sinon**

**Chercher dans la mémoire principale (RAM)**

 **Si la donnée est trouvée dans le cache → cache hit**

 **Sinon → cache miss (délai d'attente plus long)**

# Niveaux de cache, vitesse et taille

## A RETENIR SUR LES CACHES :

- Les caches sont fixes en taille, non extensibles.
- Une version de processeur avec plus de cache coûte plus cher.
- Plus un cache est proche du cœur → plus il est rapide mais moins il est volumineux.

# Accès aux registres

- L'accès aux registres est extrêmement rapide, ne limite pas la vitesse d'exécution.
- Mais très peu nombreux : chaque cœur dispose de 16 registres généraux et 16 registres SIMD.

# Accès au cache L1

- Très rapide, bande passante de 32 octets/cycle (soit 4 nombres en double précision).
- Cela suffit pour deux opérandes × deux opérations par cycle.
- Or, le cœur peut exécuter 4 opérations par cycle : pour atteindre les performances maximales, certains opérandes doivent rester en registre.
- Résultat : L1 permet environ 50% des performances maximales.

# Accès aux caches L2 et L3

Même bande passante théorique que L1, mais ralentis par les mécanismes de cohérence entre caches (synchronisation entre cœurs).

# Performances mémoire et limites des registres et caches

## Mémoire principale

- Latence très élevée : > 100 cycles.
- Bande passante ~4,5 octets/cycle, soit 7 fois moins que L1.
- Ce débit est partagé entre tous les cœurs du processeur.
- Dans les systèmes à plusieurs processeurs (2 ou 4 sockets par nœud), une partie du débit est encore perdue pour maintenir la cohérence des caches, réduisant le débit effectif par cœur.

# Erreurs de cache (cache miss)

On reconnaît généralement 3 types d'erreurs de cache

- Erreur obligatoire (Compulsory miss)
- Erreur de capacité
- Erreur de conflit

# Erreurs de cache

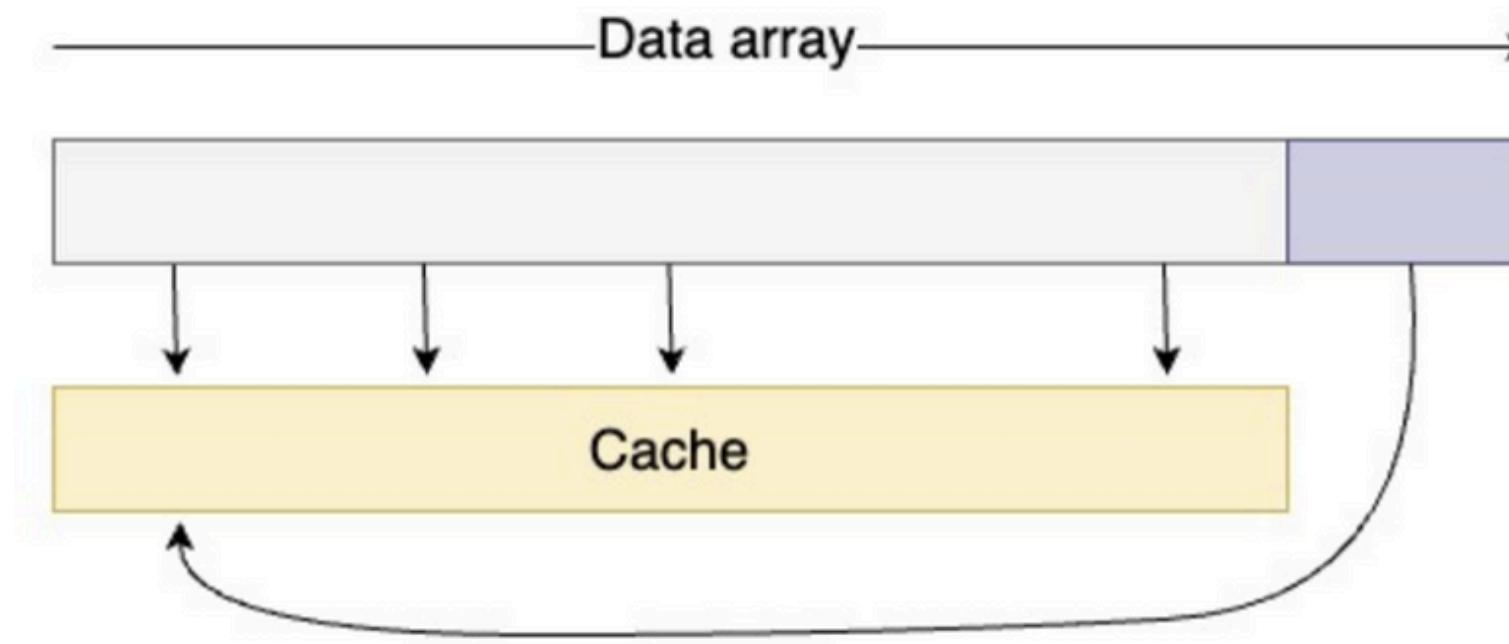
## Erreur obligatoire (Compulsory miss)

- Se produit lors du premier accès à une donnée.
- Inévitable : la donnée n'est pas encore en cache.
- Peut être atténué par le préchargement matériel (prefetching).

# Erreurs de cache

## Erreur de capacité (Capacity miss)

- Le cache est trop petit pour contenir tout l'ensemble de données actif.
- Certaines données sont évincées trop tôt car la capacité est insuffisante.
- 💡 Solution : découper le problème en blocs plus petits qui tiennent en cache.

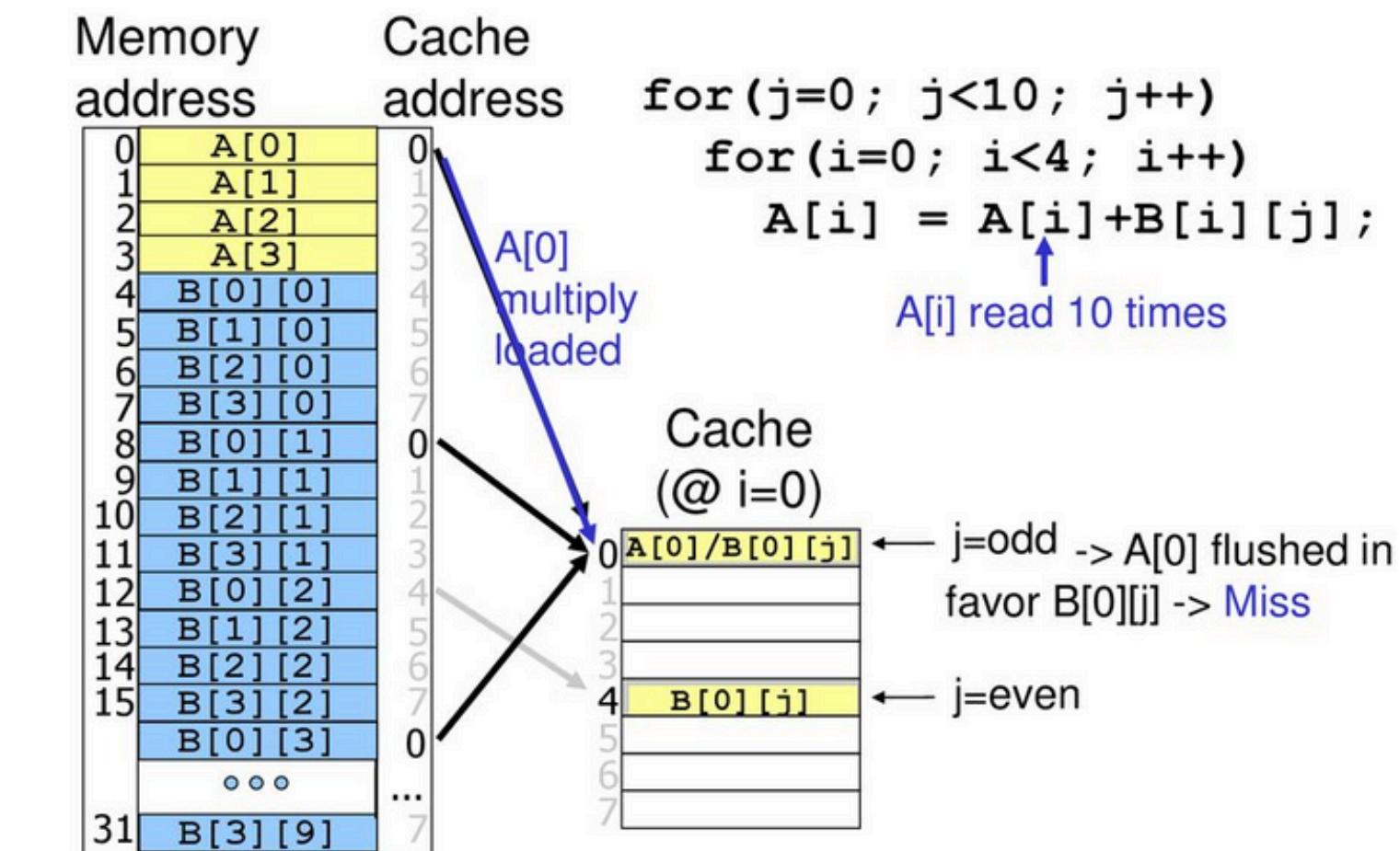


# Erreurs de cache

## Erreur de conflit (Conflict miss)

- Deux (ou plusieurs) données utilisées en même temps sont placées au même endroit dans le cache (même ligne).
- Le cache doit en évincer une, même s'il reste de la place ailleurs.
-  Cause : organisation en cache associatif à nombre de voies limité (ex: 2-way, 4-way...).

### Conflict miss example



# Erreurs de cache

## Erreur d'invalidation (en multicoeurs)

- Se produit lorsqu'un autre cœur modifie une donnée en mémoire partagée.
- La version de cette donnée dans le cache local devient alors invalide.
- Le cœur concerné doit recharger la donnée depuis la mémoire ou un autre cache valide.
- Cela fait partie de la gestion de la cohérence des caches dans les architectures multicœurs (protocole MESI, etc.).

# Stratégies de remplacement dans les caches

**Principe général :**

- Le placement et le remplacement des données en cache sont automatiquement gérés par le système (pas par le programmeur).
- Quand le cache est plein, il faut évincer une donnée pour en insérer une nouvelle.

 **Politique la plus courante : LRU (Least Recently Used)**

- On évince la donnée la moins récemment utilisée.
- ➤ Supposition : si elle n'a pas été utilisée depuis longtemps, elle est probablement moins utile.

# Stratégies de remplacement dans les caches

## Autres politiques possibles :

Stratégie	Description
FIFO	First-In First-Out : on évincé la plus ancienne entrée, peu importe son usage.
Aléatoire	Un élément est évincé au hasard, sans critère d'accès.
LFU (moins courant)	Least Frequently Used : on évincé la donnée la moins utilisée globalement.



### Remarque

- Le choix de la politique de remplacement a un impact fort sur les performances.
- LRU est généralement préférable car elle réduit les cache miss dans les programmes avec accès localisé aux données.

# Lignes de cache: Unités de transfert mémoire vers cache

Une ligne de cache (ou cache block ou cache line) est la plus petite unité de transfert entre :

- la mémoire principale et le cache,
- ou entre deux niveaux de cache.
- Taille typique : 64 à 128 octets, soit 8 à 16 nombres en double précision.



Pourquoi utiliser des lignes de cache ?

# Lignes de cache: Unités de transfert mémoire vers cache

## 1. Simplification matérielle :

- Moins de bits à gérer pour l'adressage (ex : 64 octets → 6 bits de moins).

## 2. Localité spatiale :

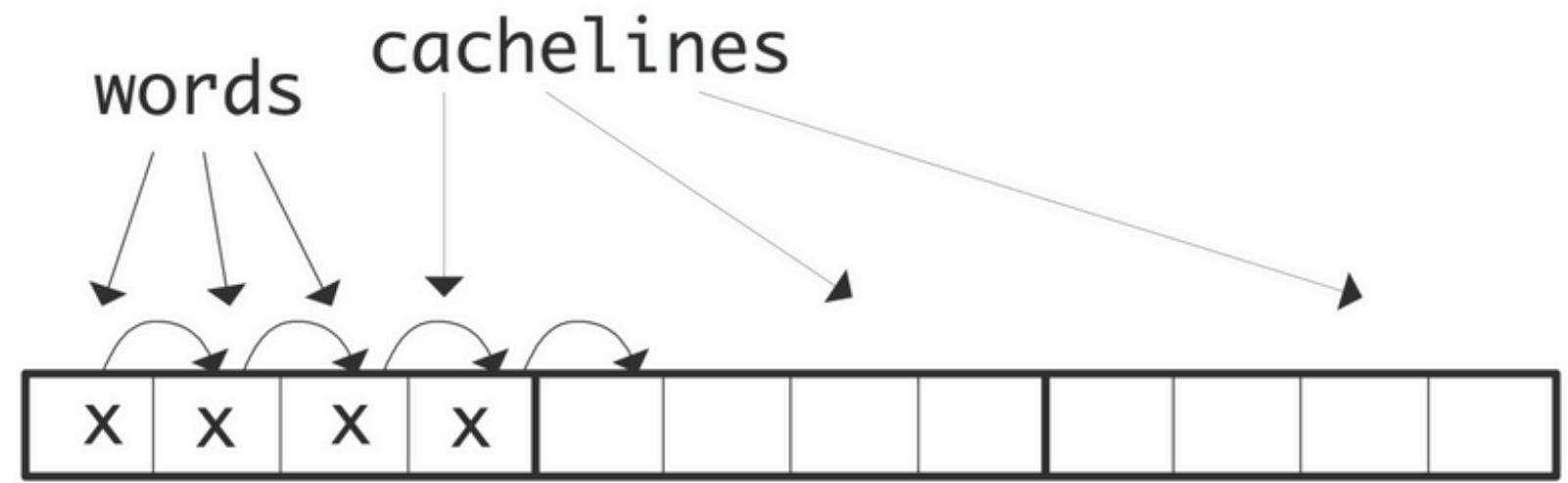
- Si un mot est utilisé, les voisins le seront probablement aussi.
- Un accès mémoire entraîne le chargement de plusieurs mots contigus.

## ⚡ Implication pour le code :

- Un bon programme exploite le contenu entier d'une ligne de cache.
- Cela encourage l'écriture de boucles avec accès contigus (stride 1).
- Accéder à d'autres éléments dans la même ligne de cache est quasiment gratuit.

# Lignes de cache: unités de transfert mémoire vers cache

Prenons comme illustration un cas avec 4 mots par ligne de cache. La demande des premiers éléments charge l'ensemble de la ligne de cache qui les contient dans la mémoire cache. Une demande pour les 2e, 3e et 4e éléments peut alors être satisfaite à partir de la mémoire cache.

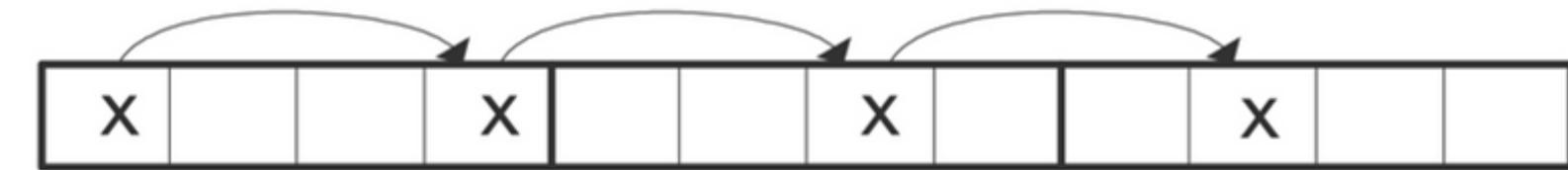


**Accès à 4 éléments en une étape**

# Lignes de cache: unités de transfert mémoire vers cache

Une plus grande ligne implique que dans chaque ligne de cache, seuls certains éléments sont utilisés.

Nous l'illustrons avec stride 3 : la demande des premiers éléments charge une ligne de cache, et cette ligne de cache contient également le deuxième élément.

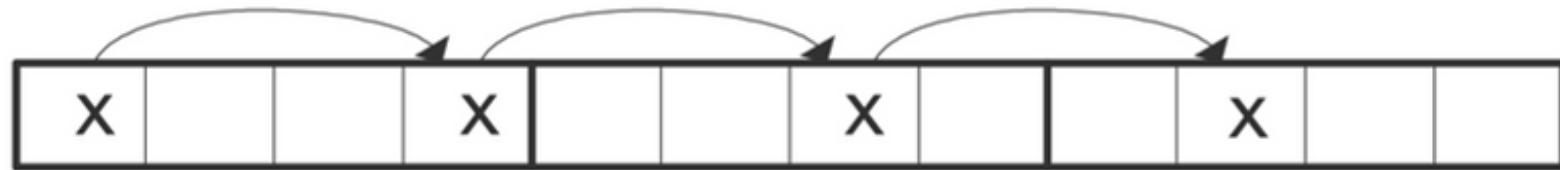


**Accès à 4 éléments en 3 étapes**

Cependant, le troisième élément se trouve sur la ligne de cache suivante, de sorte que le chargement de cette ligne implique la latence et la largeur de bande de la mémoire principale.

# Lignes de cache: unités de transfert mémoire vers cache

Il en va de même pour le quatrième élément. Le chargement de quatre éléments nécessite maintenant le chargement de trois lignes de cache au lieu d'une, ce qui signifie que les deux tiers de la bande passante disponible ont été gaspillés.



## **Accès à 4 éléments en 3 étapes**

Ce deuxième cas entraînerait également un temps de latence trois fois supérieur au premier, si n'existaient pas un mécanisme matériel qui remarque les schémas d'accès réguliers et charge de manière préemptive d'autres lignes de cache supplémentaires.

# Caches associatifs – Réduire les conflits de cache

## Problème des conflits :

- Dans un cache direct-mappé, chaque adresse mémoire ne peut aller qu'à un seul emplacement.
- Si deux données utilisées en même temps sont mappées au même emplacement, cela crée un conflit → éviction prématuée.

## Solution : Cache associatif



Exemple :

- Pour éviter l'éviction prématuée montrée précédemment, un cache à 2 voies suffit.
- En pratique, les processeurs modernes utilisent souvent des caches à 4, 8 ou 16 voies.

Type	Description
<input checked="" type="checkbox"/> Associatif complet ( <i>Fully associative</i> )	Une donnée peut aller n'importe où dans le cache → zéro conflit... mais très lent et coûteux.
<input checked="" type="checkbox"/> Cache associatif à k voies ( <i>k-way set-associative</i> )	Une donnée peut aller à l'un des k emplacements possibles d'un même groupe (set). Ex : k = 2 ou 4.

**Calcul  
parallèle**

# Calcul scientifique et parallélisation

## CONTEXTE

- Les codes scientifiques impliquent souvent un grand nombre d'opérations.
- Ces opérations sont souvent régulières : la même tâche est répétée sur un grand volume de données.
- Objectif : accélérer les calculs à l'aide de plusieurs processeurs.

### PROBLÈME DE BASE

Si l'on a  $N$  opérations à effectuer, qui prennent un temps  $T$  sur un seul processeur, peuvent-elles être effectuées en  $T/P$  avec  $P$  processeurs ?

# Addition de vecteurs

## ✓ Exemple simple : Addition de deux vecteurs

Soit deux vecteurs :

$$A = (a_1, a_2, \dots, a_n), \quad B = (b_1, b_2, \dots, b_n)$$

Opération à effectuer :  $C = A + B = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$

- Chaque addition est indépendante.
- L'addition est donc parfaitement parallélisable.
- Avec  $p$  processeurs, on peut théoriquement atteindre un temps d'exécution de  $t/p$ .

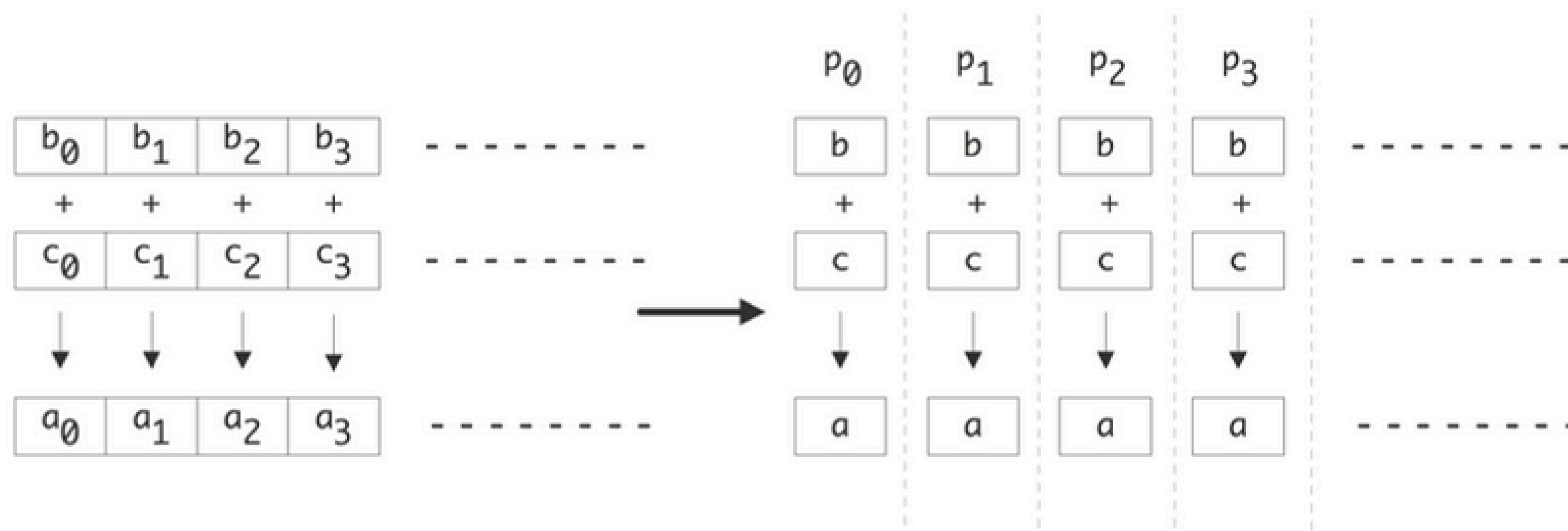
# Addition de vecteurs: schématisation

✓ Exemple simple : Addition de deux vecteurs

Soit deux vecteurs :  $A = (a_1, a_2, \dots, a_n), B = (b_1, b_2, \dots, b_n)$

Opération à effectuer :  $C = A + B = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$

## Schématisation de l'addition de deux vecteurs



# Addition de vecteurs: cas idéaliste

 **Exemple simple : Addition de deux vecteurs**

**Soit deux vecteurs :**  $A = (a_1, a_2, \dots, a_n), \quad B = (b_1, b_2, \dots, b_n)$

**Opération à effectuer :**  $C = A + B = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$

- Chaque addition est indépendante.
- L'addition est donc parfaitement parallélisable.
- Avec  $p$  processeurs, on peut théoriquement atteindre un temps d'exécution de  $t/p$ .

Cas idéaliste : accélération linéaire idéale.

Cas réaliste : communications + synchronisation + répartition des tâches

# Hypothèse irréaliste sur nombre de processeurs

L'exemple de la sommation repose sur l'hypothèse irréaliste que chaque processeur ne stocke initialement qu'un seul élément vectoriel : dans la pratique, nous aurons un nombre de processeurs ( $P$ ) plus petit que la taille des vecteurs ( $N$ ) qui peut être assez grand.

# Addition de vecteurs: cas réaliste

✓ Exemple simple : Addition de deux vecteurs avec  $P$  processeurs et  $P < N$

Soit deux vecteurs :  $A = (a_1, a_2, \dots, a_n), B = (b_1, b_2, \dots, b_n)$

Opération à effectuer :  $C = A + B = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$

**Pseudo-code pour chaque processeur**

```
for (i=my_low; i<my_high; i++)
    a[i] = b[i] + c[i];
```

# Sommation des éléments d'un vecteur

**Problème:** Sommation des éléments d'un vecteur.

**Entrée:** vecteur

**Sortie :** scalaire

Nous supposons à nouveau que chaque processeur ne contient qu'un seul élément du tableau.

Code séquentiel :    `s = 0;`  
                      `for (i=0; i<n; i++)`  
                       `s += x[i]`

Ce code n'est pas parallèle. Mais nous pouvons le réécrire pour qu'il soit parallèle.

# Sommation des éléments d'un vecteur

**Problème:** Sommation des éléments d'un vecteur.

**Entrée:** vecteur

**Sortie :** scalaire

Nous supposons à nouveau que chaque processeur ne contient qu'un seul élément du tableau.

En réécrivant la boucle comme suit, nous obtenons un code parallèle :

```
for (s=2; s<2*n; s*=2)
    for (i=0; i<n-s/2; i+=s)
        x[i] += x[i+s/2]
```

# Sommation des éléments d'un vecteur

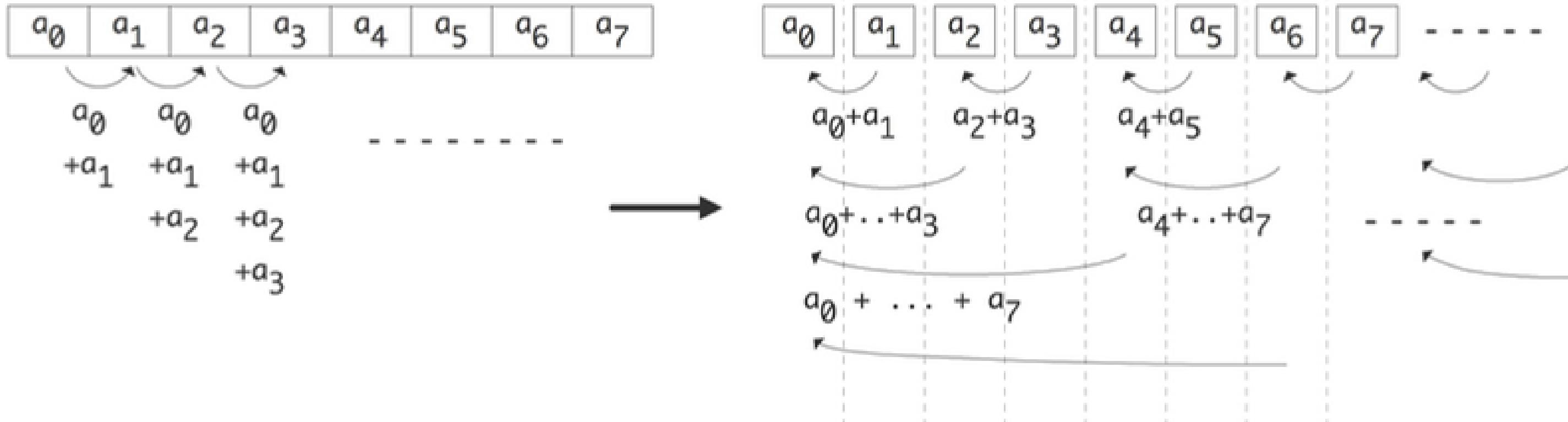
Pseudo-code parallèle

```
for (s=2; s<2*n; s*=2)
    for (i=0; i<n-s/2; i+=s)
        x[i] += x[i+s/2]
```

Réduction en  $n/p.\log_2(n)$  étapes

Utilise  $n/2$  additions à la première étape, puis moitié moins à chaque fois  
 $x[0]$  contient le résultat final

## Schématisation de la sommation parallèle



# Sommation et communication

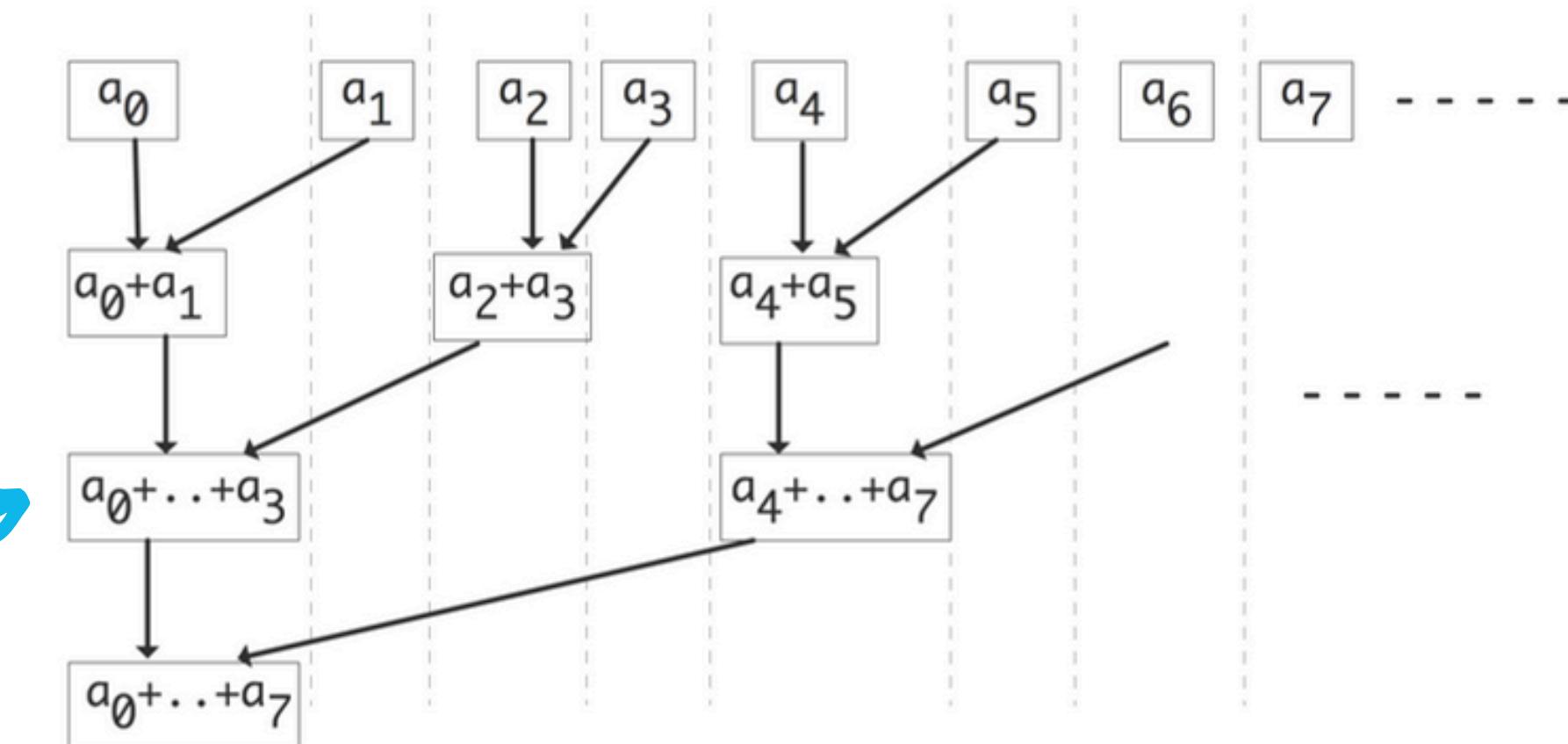
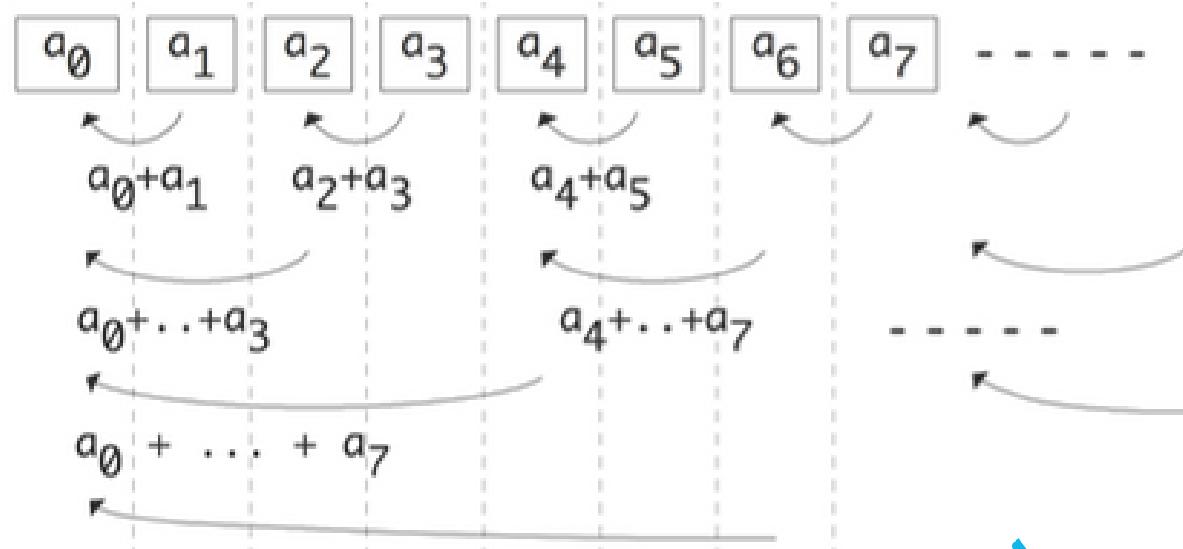
## Pseudo-code parallèle

```
for (s=2; s<2*n; s*=2)
    for (i=0; i<n-s/2; i+=s)
        x[i] += x[i+s/2]
```

## Problèmes de communication

- les processeurs sont souvent reliés par un réseau, et le déplacement des données à travers ce réseau prend du temps.

## Graphe de communication et analyse



# Exercices d'application

Supposons qu'une addition prenne une certaine unité de temps ( $k$ ), et que déplacer (envoyer) un nombre d'un processeur à un autre prenne également cette même unité de temps ( $k$ ).

**Montrez que le temps de communication est égal au temps de calcul.**

Supposons maintenant que l'envoi d'un nombre du processeur  $p$  vers le processeur  $p \pm k$  prenne un temps égal à  $k$ .

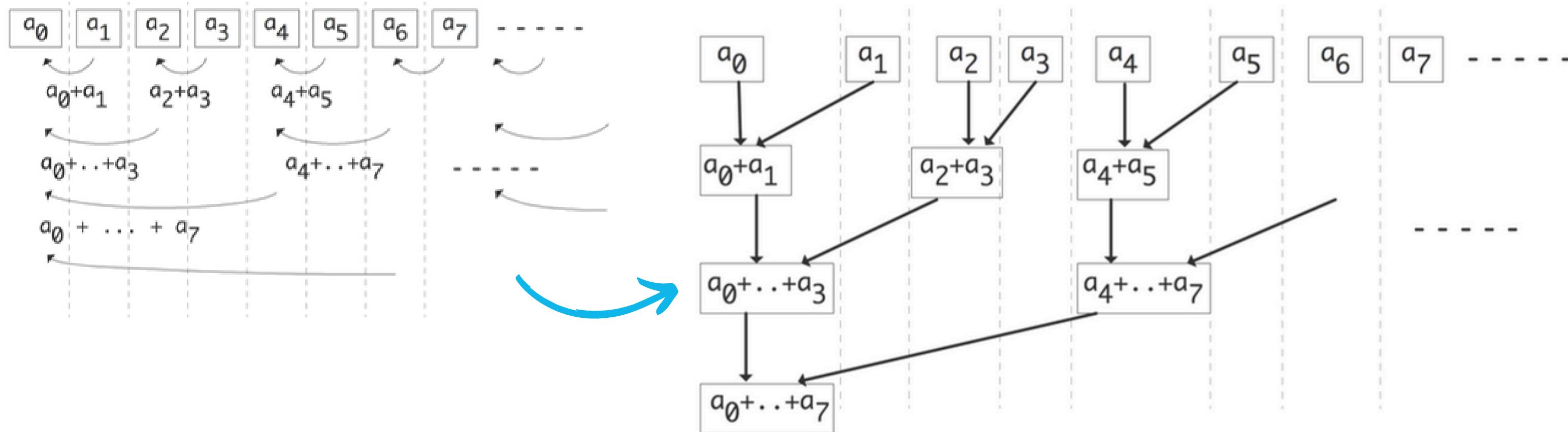
**Montrez que le temps d'exécution de l'algorithme parallèle est alors du même ordre que celui du temps séquentiel.**

# Exercices d'application

Considérons le cas de la somme de 8 éléments à l'aide de 4 processeurs.  
**Montrez que certaines arêtes dans le graphe de communication du problème de sommation ne correspondent plus à des communications réelles.**

Considérons maintenant la somme de 16 éléments, toujours avec 4 processeurs.

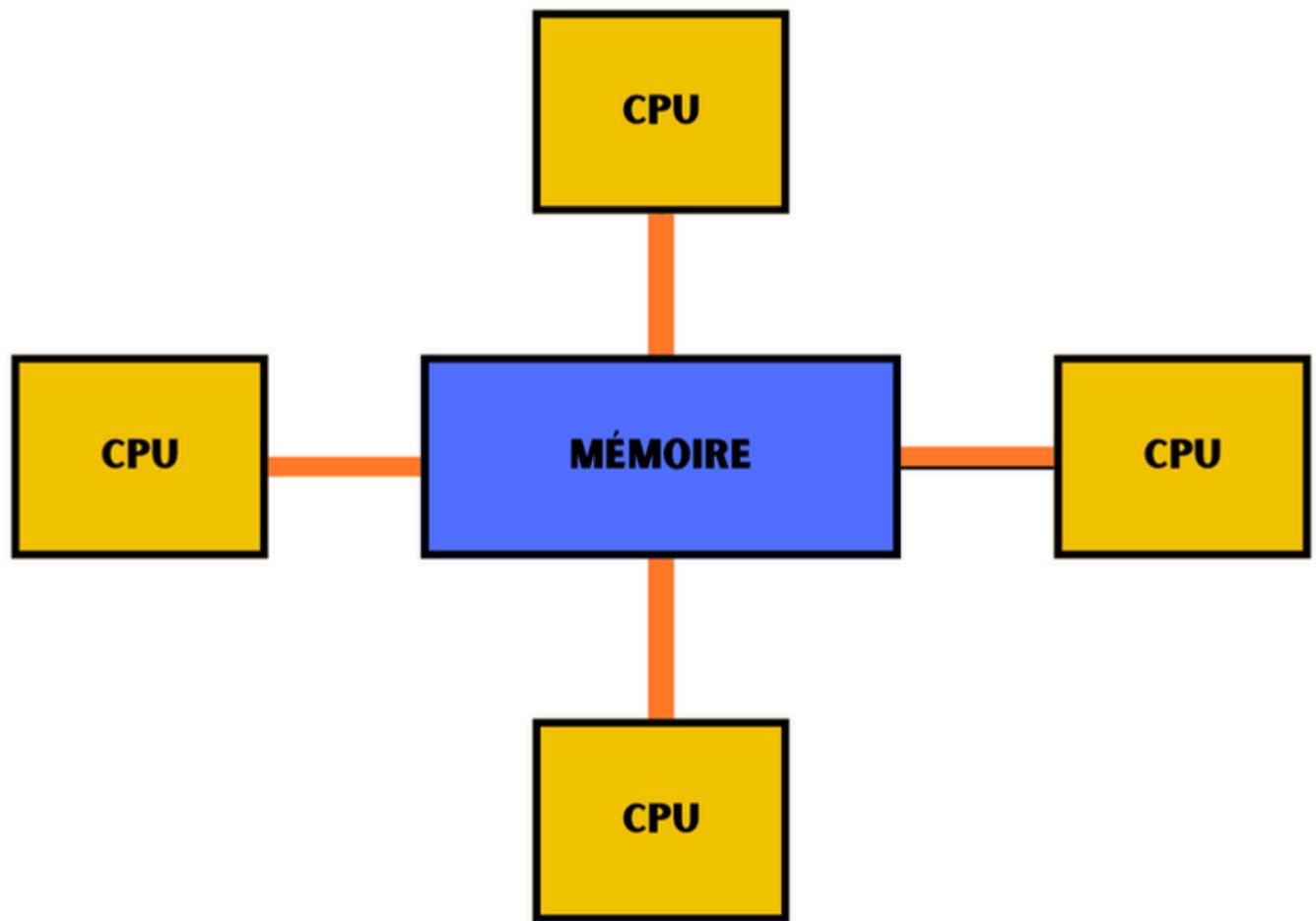
**Quel est le nombre d'arêtes de communication cette fois ?**



# Mémoire et accès dans les machines parallèles

# Mémoire Uniforme (UMA)

- Tout processeur peut accéder à n'importe quelle zone mémoire avec le même temps d'accès
- Ce modèle facilite la programmation parallèle
- Appelé aussi SMP (Symmetric Multi-Processing)



Accès mémoire uniforme (SMP)

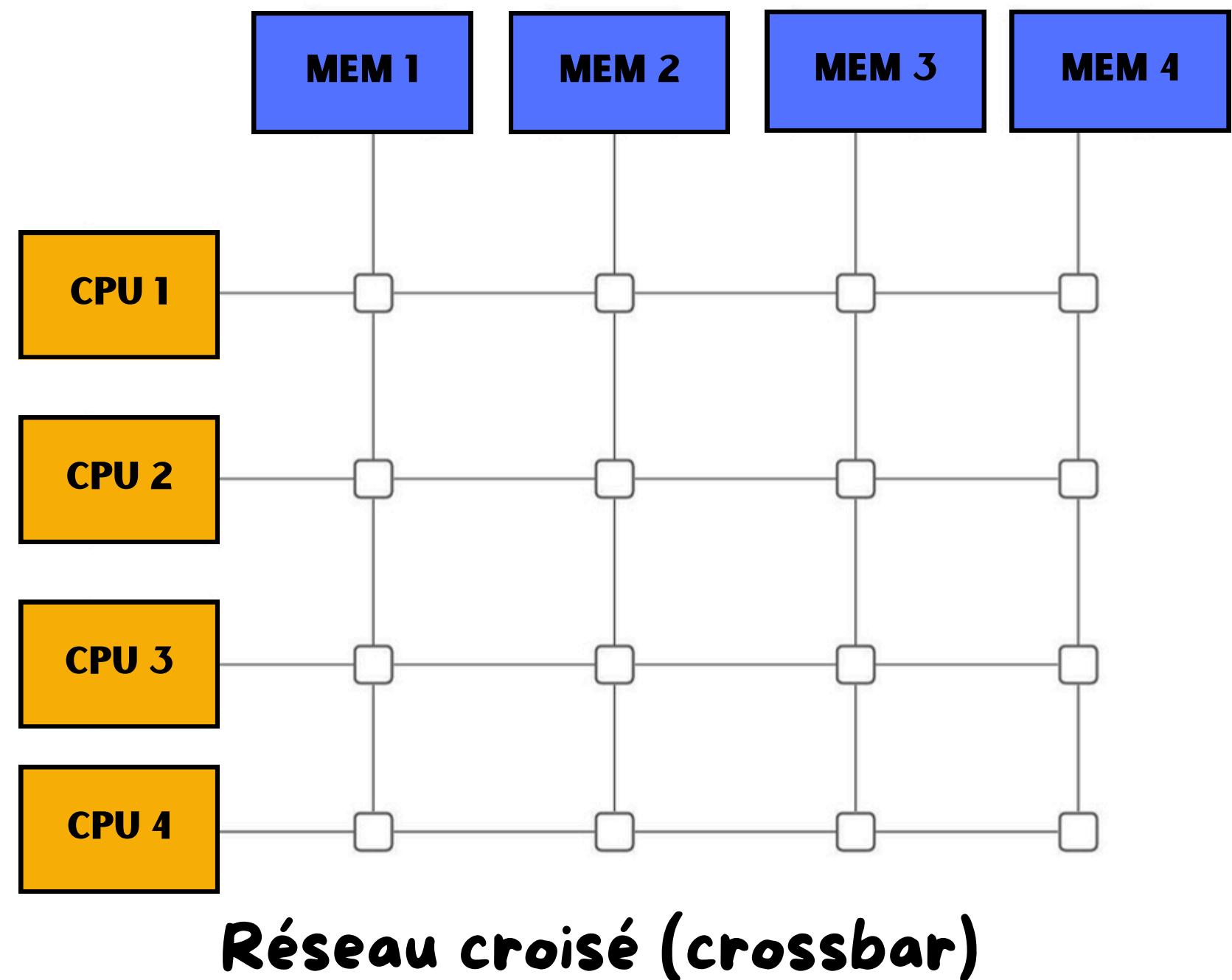
# Mémoire Uniforme (UMA)

Réalisation de l'architecture SMP

- Les processeurs partagent la mémoire via un bus mémoire unique.
  - Ex. : Mac avec 2 processeurs (sockets) à 6 cœurs
- Limité à un petit nombre de processeurs

Pour augmenter le nombre de processeurs :

- On utilise un réseau croisé (crossbar) pour relier plusieurs processeurs à plusieurs banques mémoire



# Mémoire Uniforme (UMA)

Cas des multicœurs :

- Les cœurs partagent souvent une mémoire cache (L2 ou L3)
- Forme spéciale d'accès mémoire uniforme

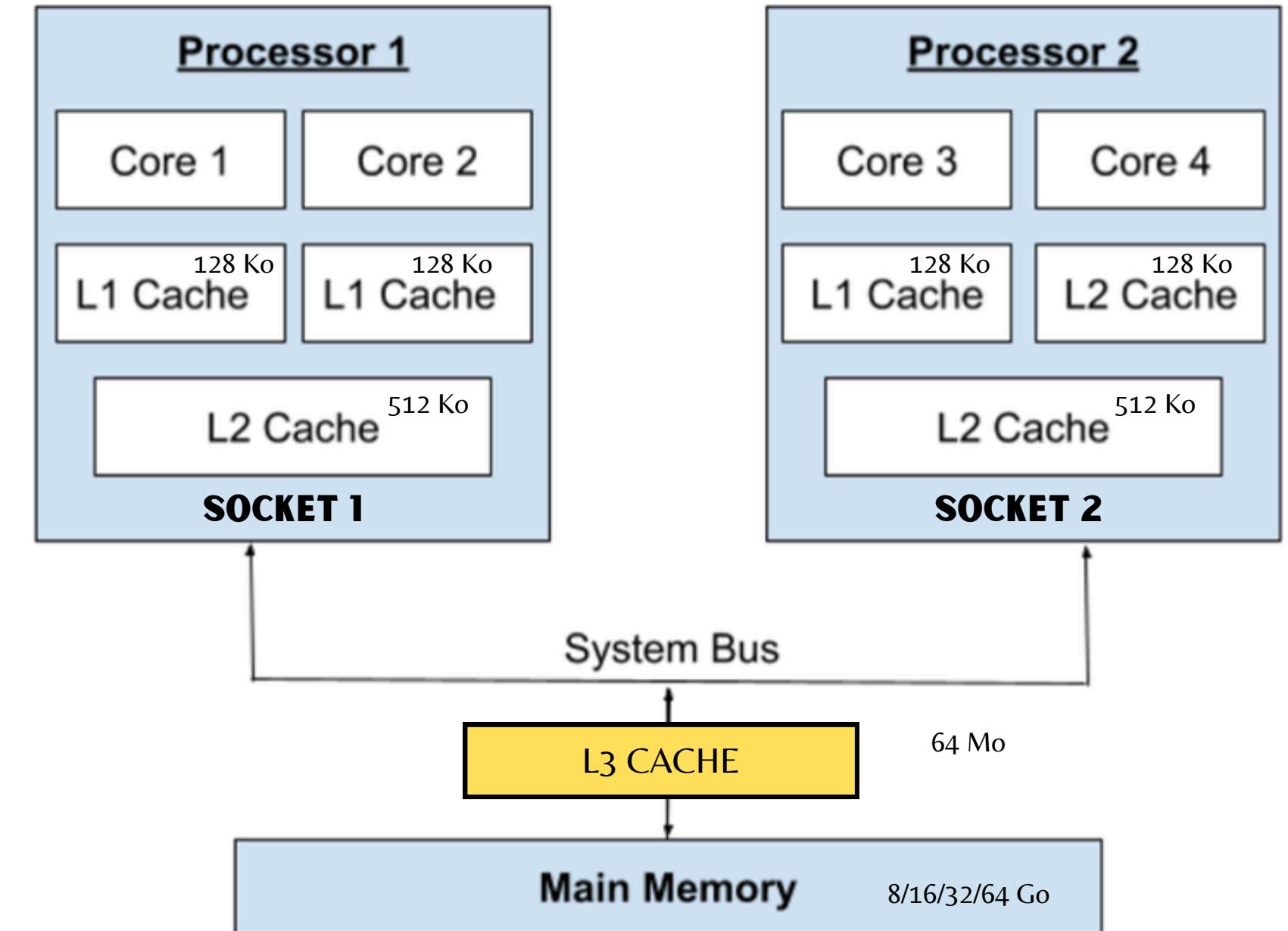
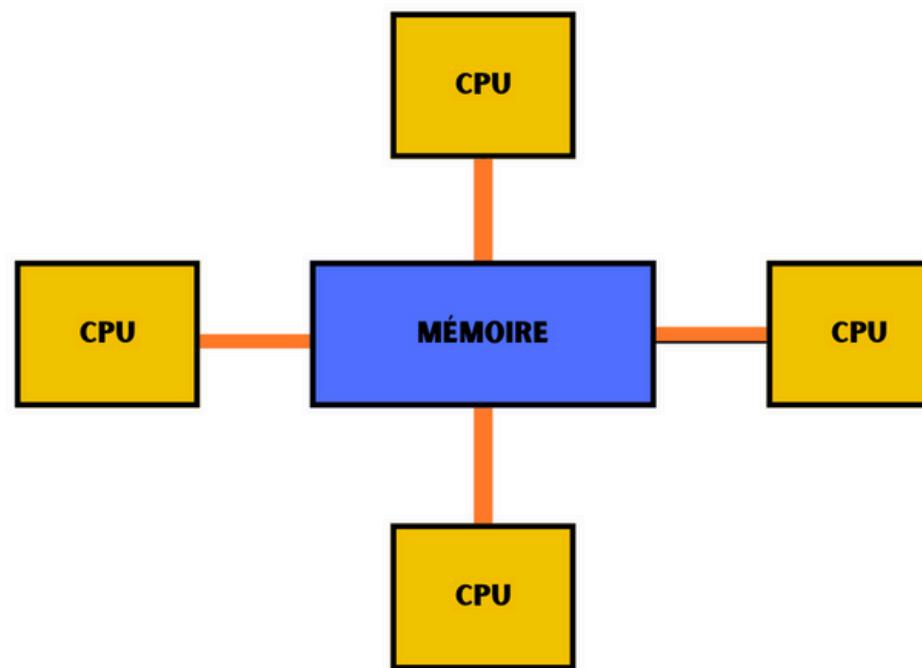


Fig. 2: Multi-core Processor Architecture

## Accès mémoire uniforme (SMP)

# Mémoire Uniforme (UMA) : limites et contraintes

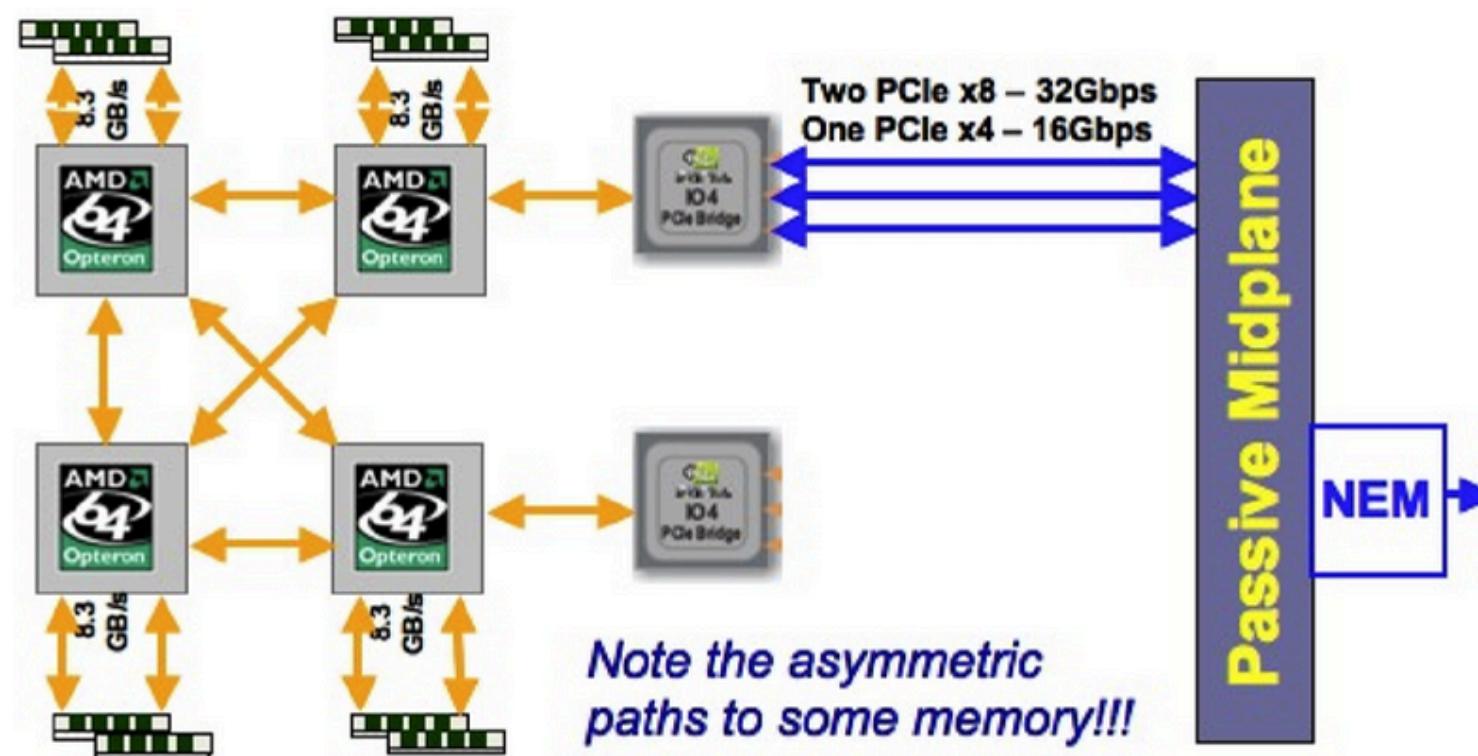
- Chaque processeur accède à une mémoire partagée via un bus ou une interconnexion commune.
- **Cette approche fonctionne bien pour un petit nombre de processeurs.**
- **Mais au-delà de 10 CPU, la bande passante devient insuffisante.**
- Les temps d'accès augmentent fortement, provoquant un goulot d'étranglement qui limite les performances globales.
- Difficulté à scaler efficacement pour les systèmes massivement parallèles.

# Accès mémoire non uniforme (NUMA): exemple

## Solution NUMA

- Chaque processeur a sa mémoire locale rapide.
- Il peut accéder à la mémoire des autres, mais plus lentement.
- On abandonne l'uniformité, mais on garde l'espace mémoire logique partagé.

**Non-uniform memory access in a four-socket motherboard.**

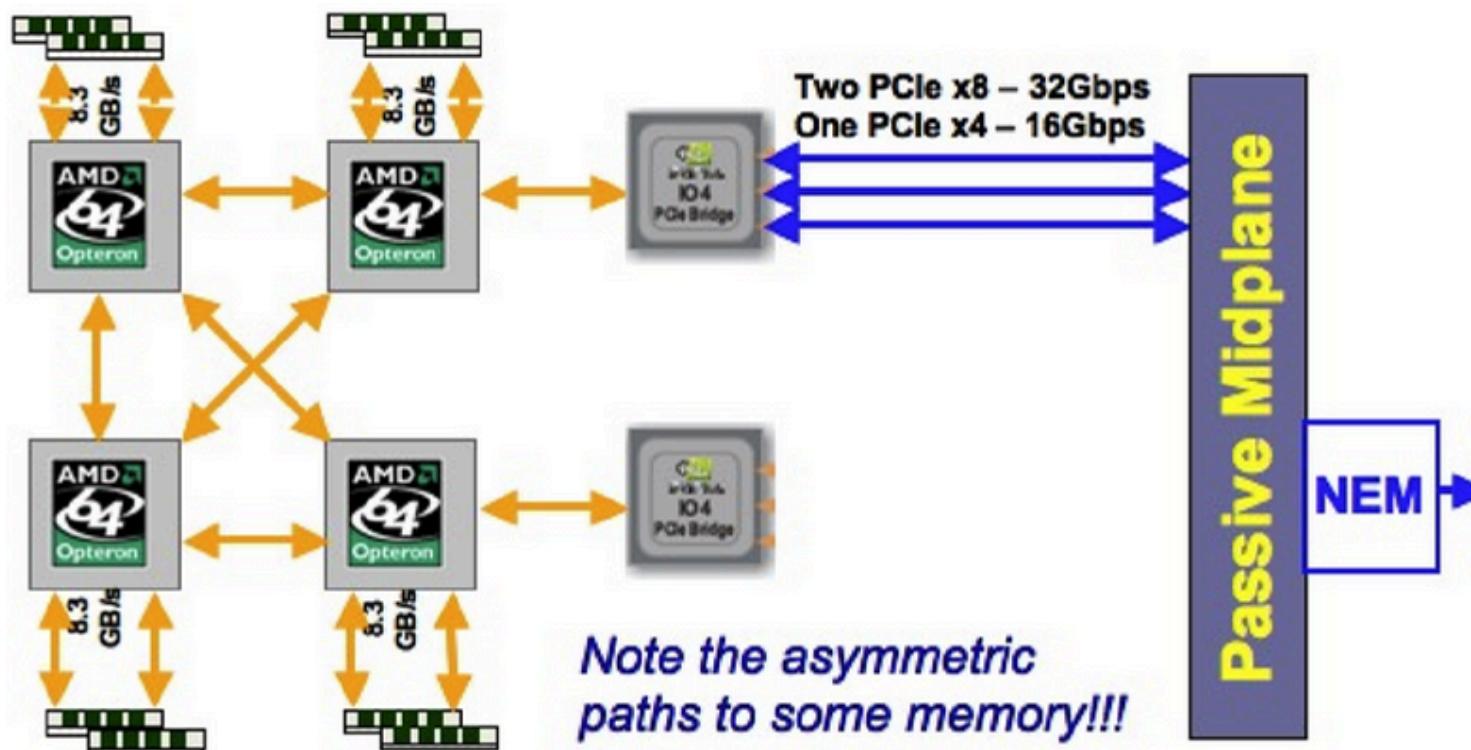


# Accès mémoire non uniforme (NUMA): exemple

Architecture NUMA avec 4 processeurs sur le cluster TACC Ranger

- 8 Go de mémoire locale, illusion d'une mémoire partagée de 32 Go.
- Accéder à sa propre mémoire est rapide
- Accéder à celle des autres est plus lent.
- Deux processeurs doivent passer par un processeur intermédiaire pour accéder à la mémoire des autres, ce qui ralentit les transferts et sature les connexions.

**Non-uniform memory access in a four-socket motherboard.**

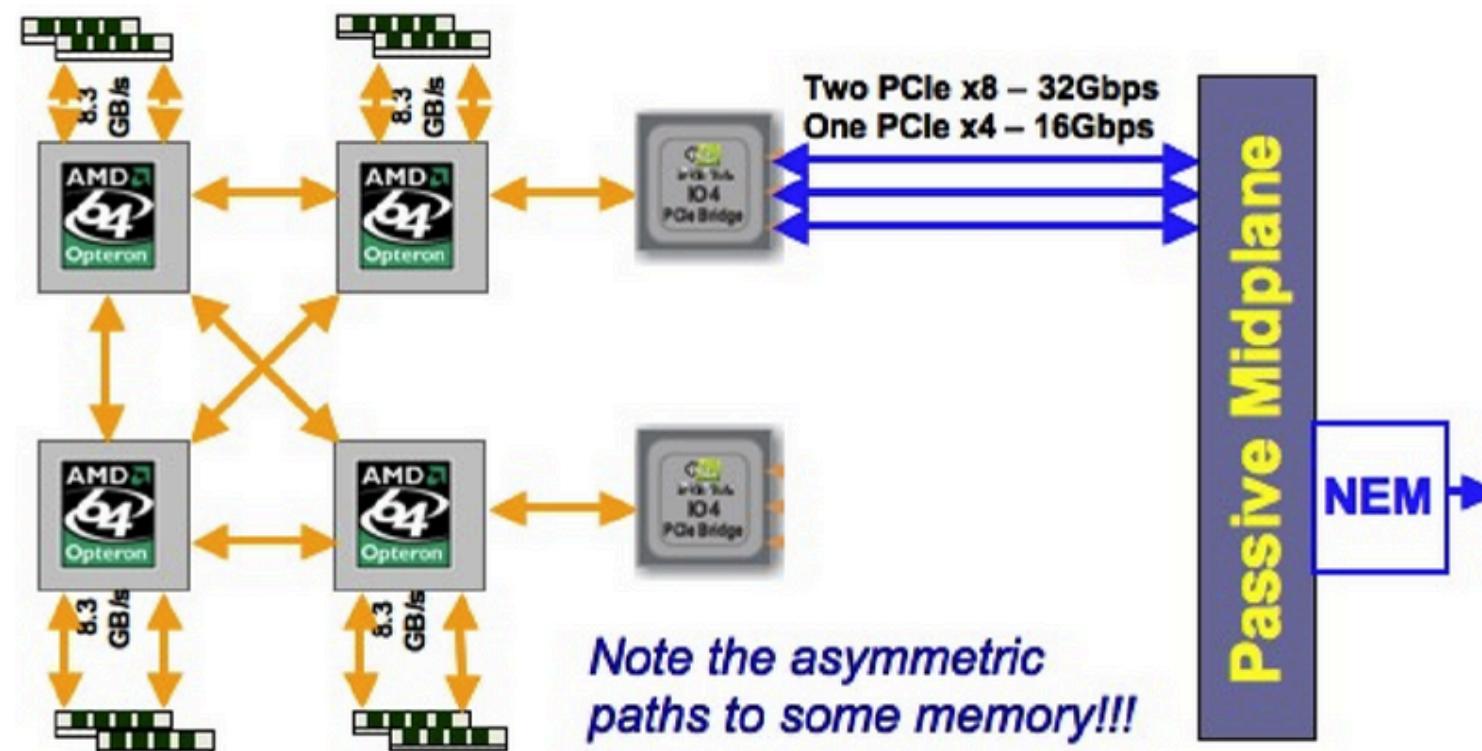


# Accès mémoire non uniforme (NUMA)

## Affinité mémoire (affinity).

- La localisation des données par rapport aux coeurs est cruciale.
- On parle d'affinité mémoire ou affinité de processus.
- Ex. : TACC Ranger (4 processeurs avec 8 Go chacun) → 32 Go logiques, mais accès non symétrique.

**Non-uniform memory access in a four-socket motherboard.**



# NUMA et gestion de caches

## Cohérence du cache (cache coherence)

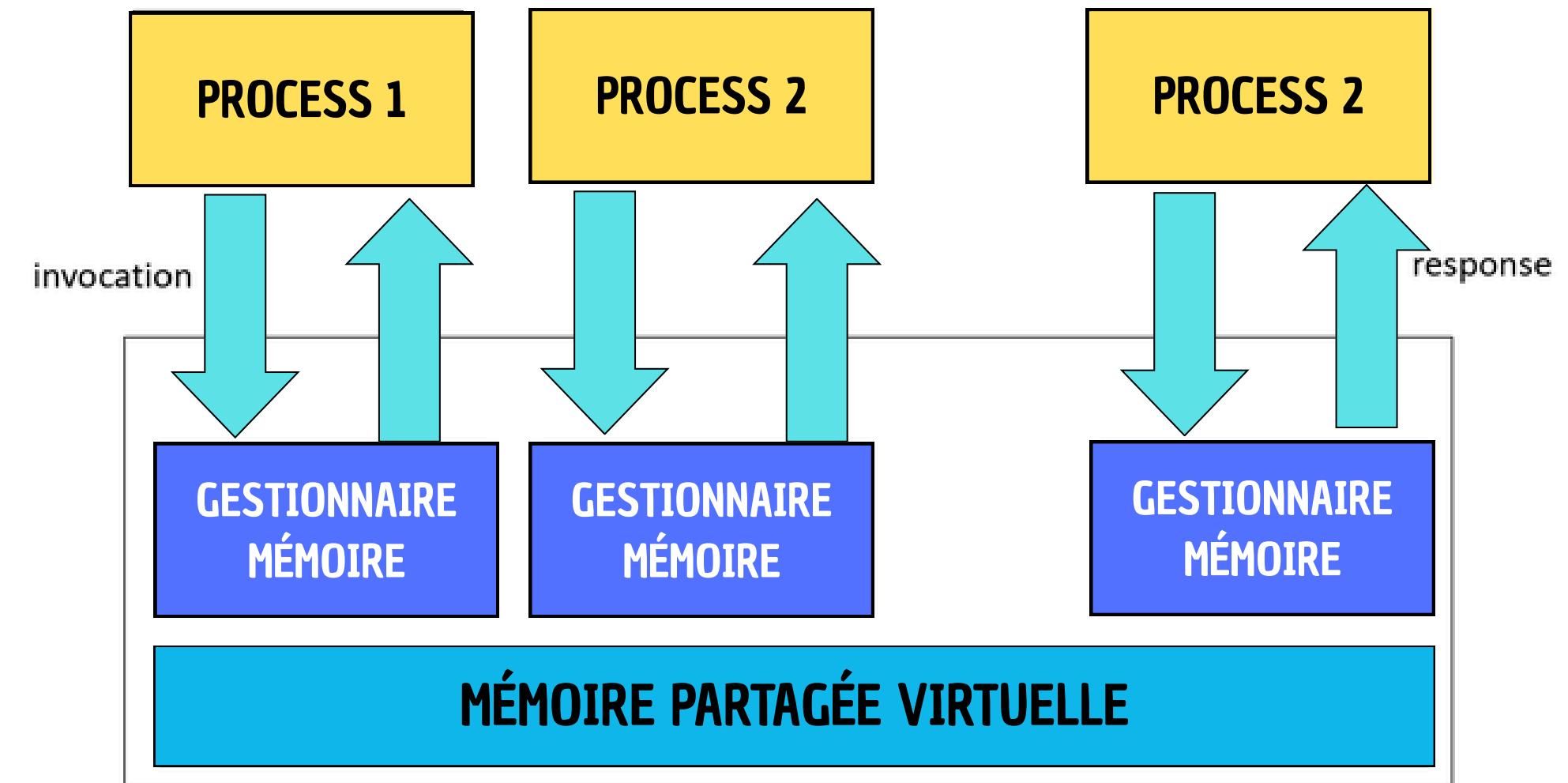
- Les processeurs conservent parfois une copie locale d'une même donnée.
- Si l'un la modifie, tous les caches doivent être synchronisés.
- Cela pose des défis système → nécessite une architecture ccNUMA (cache-coherent NUMA)

# Virtualisation d'une mémoire partagée

## Mémoire partagée virtuelle

(Distributed Shared Memory)

- Une couche logicielle peut faire apparaître un cluster comme une mémoire partagée.
- Exemple : ScaleMP, SGI UV, Cray XE6.
- Utilisation de langages PGAS (Partitioned Global Address Space)
- Le noyau Linux peut gérer jusqu'à 4096 threads avec ce modèle.



# **Instruction-Level Parallelism (ILP)**

# Instruction-Level Parallelism (ILP)

L'Instruction-Level Parallelism consiste à exécuter plusieurs instructions en parallèle à l'intérieur d'un seul cœur de processeur, tant qu'elles ne dépendent pas les unes des autres.

Exemple simple:

- $a = b + c;$
- $d = e * f;$
- $g = h - i;$
- $j = a + d;$

## Étape 1 — Analyse des dépendances

Instruction	Dépend-elle d'une autre ?	Peut-elle être exécutée en parallèle ?
$a = b + c$	Non	Oui
$d = e * f$	Non	Oui
$g = h - i$	Non	Oui
$j = a + d$	Oui (dépend de a et d)	Non (doit attendre a et d)

# Instruction-Level Parallelism (ILP)

L'Instruction-Level Parallelism consiste à exécuter plusieurs instructions en parallèle à l'intérieur d'un seul cœur de processeur, tant qu'elles ne dépendent pas les unes des autres.

Exemple simple:

- $a = b + c;$
- $d = e * f;$
- $g = h - i;$
- $j = a + d;$

Étape 2 — Parallélisme possible



Cycle 1

Exécuter en parallèle :

$a = b + c$   
 $d = e * f$   
 $g = h - i$



Cycle 2

Exécute :

$j = a + d$  (maintenant que  
a et d sont prêts)

# Rôle de l'ILP

Dans l'exemple précédent, un processeur superscalaire avec 3 unités d'exécution peut :

- détecter qu'aucune dépendance ne lie les calculs de a, d, et g
- les envoyer en parallèle à trois unités différentes
- ensuite, il attend que a et d soient prêts pour faire j

## Pourquoi c'est important ?

- ILP permet à un cœur de faire plus d'opérations par cycle.
- Il est automatiquement exploité par le matériel (processeur + compilateur).
- Moins ton code a de dépendances, mieux le processeur peut le paralléliser.

# ILP en assembleur

LOAD R1, A	; Charger A dans registre R1
LOAD R2, B	; Charger B dans R2
ADD R3, R1, R2	; $R3 = A + B$
LOAD R4, C	; Charger C dans R4
MUL R5, R3, R4	; $R5 = (A + B) * C$

## ILP ?

- Les 2 premières instructions LOAD sont indépendantes → parallélisables.
- ADD dépend des LOAD → doit attendre.
- LOAD R4, C peut se faire en même temps que le ADD → parallèle.
- MUL attend le résultat du ADD → dépendant.

# ILP en assembleur

```
1. mov eax, [a] ; charger a  
2. mov ebx, [b] ; charger b  
3. add eax, ebx ; x = a + b  
4. mov ecx, [c] ; charger c  
5. mov edx, [d] ; charger d  
6. mul ecx, edx ; y = c * d  
7. sub eax, ecx ; z = x - y
```

On veut calculer  
 $x = a + b;$   
 $y = c * d;$   
 $z = x - y;$

Cycle 1: | mov eax, [a] | mov ebx, [b] | mov ecx, [c] | mov edx, [d]

Cycle 2: | add eax, ebx | imul ecx, edx

Cycle 3: | sub eax, ecx

# Outils de monitoring ILP



Visiter <https://godbolt.org/>  
pour visualiser le profil de votre code ILP

The screenshot shows the Compiler Explorer interface. On the left, the C source code for a `compute` function is displayed:

```
1 int compute(int a, int b, int c, int d) {
2     int x = a + b;
3     int y = c * d;
4     int z = x - y;
5     return z;
6 }
```

The assembly output on the right is generated by x86-64 gcc 15.1. The assembly code corresponds to the source code, showing the stack frame setup, variable assignments, and the final subtraction and return operation.

Line	Assembly Instruction
1	compute:
2	push rbp
3	mov rbp, rsp
4	mov DWORD PTR [rbp-20], edi
5	mov DWORD PTR [rbp-24], esi
6	mov DWORD PTR [rbp-28], edx
7	mov DWORD PTR [rbp-32], ecx
8	mov edx, DWORD PTR [rbp-20]
9	mov eax, DWORD PTR [rbp-24]
10	add eax, edx
11	mov DWORD PTR [rbp-4], eax
12	mov eax, DWORD PTR [rbp-28]
13	imul eax, DWORD PTR [rbp-32]
14	mov DWORD PTR [rbp-8], eax
15	mov eax, DWORD PTR [rbp-4]
16	sub eax, DWORD PTR [rbp-8]
17	mov DWORD PTR [rbp-12], eax
18	mov eax, DWORD PTR [rbp-12]
19	pop rbp
20	ret

# ILP vs Parallélisme des données

L'addition parallèle de deux vecteurs est un exemple de parallélisme des données.

Ci-dessous un exemple d'implémentation en OpenMP.

```
#include <omp.h>

void add_arrays(float* a, float* b, float* x, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        x[i] = a[i] + b[i];
    }
}
```

# ILP vs Parallélisme des données

Caractéristique	Instruction-Level Parallelism (ILP)	Data Parallelism
Définition	Exécution simultanée de plusieurs instructions indépendantes	Exécution simultanée de la même instruction sur plusieurs données
Qui le gère ?	Le processeur (hardware) avec pipeline, out-of-order, superscalaire	Le programmeur ou le compilateur, avec des boucles vectorisées ou du multithreading
Exemple typique	Calcul : $x = a + b ; y = c * d ; z = x - y$ (instructions indépendantes)	Appliquer la même opération à tout un tableau : $y[i] = x[i] + 2$
Niveau	Bas niveau : dans un seul cœur, sur quelques instructions	Haut niveau : sur des structures de données entières (vecteurs, tableaux)
Utilisé par	Processeurs superscalaires, out-of-order, pipelines	SIMD, OpenMP, CUDA, TensorFlow, etc.
Exécution parallèle ?	Oui, mais sur une seule instance de données	Oui, sur plusieurs données en parallèle
Granularité	Très fine (instruction)	Plus grossière (données, tableaux)
Outils / modèles	Matériel : pipelines, unités d'exécution multiples	Langages / modèles : for parallèles, OpenMP, CUDA, SIMD, etc.

# Dépendance ET Propriétés

# Dépendances et propriétés

- Les dépendances imposent des contraintes pour garantir que le programme transformé reste correct.
- Elles concernent :
  - l'Ordre de production et de consommation des données (dépendances de données)
  - le Contrôle du flux d'exécution (dépendances de contrôle)

# Dépendances et propriétés

Exemple simple:

**S1 : PI = 3.14**

**S2 : R = 5.0**

**S3 : AREA = PI \* R \*\* 2**

Propriété :

- S3 ne peut pas être exécutée avant S1 ou S2 → risque d'utiliser des valeurs non définies.
- Dépendances de données :
  - S1 → S3
  - S2 → S3

# Dépendances et propriétés

Exemple simple:

**S1 : PI = 3.14**

**S2 : R = 5.0**

**S3 : AREA = PI \* R \*\* 2**

Pas de contrainte inutile

- Ordre entre S1 et S2 :
  - S1 et S2 peuvent être échangées librement : l'ordre S2, S1, S3 produit le même résultat que S1, S2, S3.

Donc :

- Pas de dépendance de données entre S1 et S2.

# Dépendances et propriétés

Exemple :

**S1 : IF ( $T \neq 0.0$ ) GOTO S3**

**S2 :  $A = A / T$**

**S3 : CONTINUE**

Ici :

- L'exécution de S2 dépend du test fait dans S1.
- Exécuter S2 avant S1 peut provoquer une division par zéro → comportement incorrect.

✓ On a donc une dépendance de contrôle :

- $S1 \rightarrow S2$

# Dépendances : définition formelle

## Définition : Dépendance de données

Il existe une dépendance de données de S1 vers S2 si et seulement si :

1. Les deux instructions accèdent à la même adresse mémoire et au moins l'une d'elles y écrit.
2. Il existe un chemin d'exécution possible de S1 vers S2.

# Classification des dépendances

# Classification des dépendances Load-Store

Les dépendances entre instructions peuvent être décrites en termes d'ordre de lecture et d'écriture.

Trois types principaux de dépendance :

1. Dépendance vraie (True dependence)
2. Antidépendance (Antidependence)
3. Dépendance de sortie (Output dependence)

# Dépendance Vraie

Exemple :

**S1 : X = ...**

**S2 : ... = X**

- La première instruction écrit dans X ; la seconde lit X.
- La dépendance garantit que S2 utilise la valeur calculée par S1.
- Aussi appelée dépendance de flux (Flow dependence).
- Notation :  $S1 \delta S2$  ( $S2$  dépend de  $S1$ ).

 Représentation graphique : flèche du source ( $S1$ ) vers le puits ( $S2$ ).

# Antidépendance

Exemple :

**S1** : ... = X

**S2** : X = ...

- La première instruction lit X ; la seconde écrit dans X.
- L'antidépendance empêche d'échanger S1 et S2.
- Sinon, S1 pourrait lire la nouvelle valeur de X écrite par S2 — ce qui introduirait une fausse dépendance de flux.

 Notation :  $S_1 \delta^- S_2$  ou  $S_1 \delta^{-1} S_2$

# Dépendance de sortie

Exemple :

S1 : X = ...

S2 : X = ...

- Les deux instructions écrivent dans la même variable X.
- La dépendance empêche un échange qui pourrait rendre la valeur finale incorrecte.

# Dépendance de sortie

Exemple pratique :

S1 :  $X = 1$

S2 : ...

S3 :  $X = 2$

S4 :  $W = X * Y$

Exécuter S3 avant S1 pourrait conduire W à utiliser  $X = 1$  au lieu de  $X = 2$ .

 Notation :  $S1 \delta_0 S2$

# Dépendances et contraintes : résumé

Ces contraintes limitent :

- le réordonnancement des instructions
- la parallélisation
- l'optimisation du pipeline

**Bien identifier chaque type de dépendance est essentiel pour la parallélisation de code !**

Type de dépendance	Lecture/Écriture	Risque matériel
Dépendance vraie	Écriture → Lecture	RAW
Antidépendance	Lecture → Écriture	WAR
Dépendance de sortie	Écriture → Écriture	WAW

# **Formalisation, Itérations ET Ordonnancement**

# Formalisation et itération vectorielle

Étendre le concept de dépendance aux boucles suppose de paramétriser les instructions par les itérations.

Exemple :

```
DO I = 1, N
S1      A(I+1) = A(I) + B(I)
ENDDO
```

Ici, S1 dépend de l'instance précédente de S1.

# Formalisation et itération vectorielle

## Dépendance à plusieurs itérations

Une simple modification d'indice peut changer la distance de dépendance :

```
DO I = 1, N  
S1      A(I+2) = A(I) + B(I)  
ENDDO
```

Ici, S1 dépend de l'instance deux itérations plus tôt.

# Formalisation et itération vectorielle

## Numéro d'itération

- Boucle simple :
  - Le numéro d'itération = valeur de I

```
DO I = 1, N  
    ...  
ENDDO
```

- Boucle générique :

```
DO I = L, U, S  
    ...  
ENDDO
```

- Numéro d'itération normalisé :  $\text{iter} = (I - L + 1)/S$

# Boucles imbriquées

- Dans des boucles imbriquées :
  - Le niveau d'imbrication est égal au nombre de boucles qui englobent la boucle + 1.
- Exemple :
  - Plus on est à l'intérieur, plus le niveau est élevé.

# Boucles imbriquées et vecteur d'itérations

## Vecteur d'itération

Définition : Dans une imbrication de n boucles, le vecteur d'itération est exprimé par i :

$$i = \{i_1, i_2, \dots, i_n\}$$

où  $i_k$  est le numéro d'itération de la boucle au niveau k.

# Boucles imbriquées et vecteur d'itérations

## Exemple

```
DO I = 1, 2  
    DO J = 1, 2  
        S  
    ENDDO  
ENDDO
```

- L'instance  $S[(2,1)]$  = exécution à  $I=2, J=1$

## Espace d'itération

- L'ensemble de tous les vecteurs d'itérations forme l'espace d'itération
- Espace d'itération de  $S$  :  $\{(1,1), (1,2), (2,1), (2,2)\}$

# Boucles imbriquées : Ordre lexicographique

## Définition :

Soient  $i$  et  $j$  deux vecteurs d'itération de longueur  $n$ .

$i < j$  ssi :

- $i[1:n-1] < j[1:n-1]$  ou
- $i[1:n-1] = j[1:n-1]$  et  $i_n < j_n$ .

 Cette relation exprime l'ordre d'exécution des instances.

# Boucles imbriquées et dépendance

## Théorème : Dépendance de boucle

Il existe une dépendance de  $S_1$  vers  $S_2$  dans des boucles imbriquées ssi :

- $i < j$  ou  $i = j$  et  $S_1$  précède  $S_2$  dans le corps.
- $S_1$  et  $S_2$  accèdent à la même adresse mémoire.
- Au moins un des accès est une écriture.

# Boucles imbriquées et dépendance

## Exemple 01:



### Analyse :

- S1 lit  $A[i][j-1]$  et écrit  $A[i][j]$
- Il y a potentiellement accès à la même adresse mémoire à des itérations différentes.

```
for (int i = 0; i < N; i++) {  
    for (int j = 1; j < N; j++) {  
        A[i][j] = A[i][j-1] + 1; // s1  
    }  
}
```

# Boucles imbriquées et dépendance

## Exemple 01:



### Analyse :

- S1 lit  $A[i][j-1]$  et écrit  $A[i][j]$
- Il y a potentiellement accès à la même adresse mémoire à des itérations différentes.

```
for (int i = 0; i < N; i++) {  
    for (int j = 1; j < N; j++) {  
        A[i][j] = A[i][j-1] + 1; // s1  
    }  
}
```

### Vecteurs d'itération :

- Pour l'écriture à  $A[i][j]$ , l'itération est :  $I_1 = (i, j)$
- Pour la lecture de  $A[i][j-1]$ , c'est  $I_2 = (i, j-1)$

On a donc un flux de dépendance vraie :

$S_1(I_2) \rightarrow S_1(I_1)$  où  $I_2 = (i, j-1)$  et  $I_1 = (i, j)$

# Boucles imbriquées et dépendance

## Exemple 01:

### Théorème de dépendance :

- $A[i][j-1]$  dans l2 est la même adresse mémoire que  $A[i][j-1]$  dans l1
  - Satisfait condition 1 : même adresse mémoire
  - Écriture dans l1, lecture dans l2  $\rightarrow$  condition 2 : au moins un accès est une écriture
  - $l2 = (i, j-1)$  précède  $l1 = (i, j) \Rightarrow$  condition 3 :  $i == i, j-1 < j$
-  Il existe une dépendance  $S1[l2] \rightarrow S1[l1]$

```
for (int i = 0; i < N; i++) {  
    for (int j = 1; j < N; j++) {  
        A[i][j] = A[i][j-1] + 1; // S1  
    }  
}
```

# Boucles imbriquées et dépendance

## Exemple 02:



### Analyse :

- S1 lit  $B[i][j]$
- S2 écrit  $B[i][j]$
- Accès à la même adresse mémoire
- S1 précède S2 dans le corps

### Vecteurs d'itération :

- S1 :  $I_1 = (i, j)$
- S2 :  $I_2 = (i, j)$

```
for (int i = 0; i < N; i++) {  
    for (int j = 1; j < N; j++) {  
        A[i][j] = A[i][j-1] + 1; // S1  
    }  
}
```

# Boucles imbriquées et dépendance

## Exemple 02:

### Théorème de dépendance

- Même adresse mémoire :  $B[i][j]$
- Lecture dans  $S_1$ , écriture dans  $S_2 \rightarrow$  au moins un accès en écriture
- $i == i$  et  $j == j \rightarrow l_1 = l_2$ , mais  $S_1$  précède  $S_2$  dans le corps
- Il existe une dépendance  $S_1 \rightarrow S_2$
- Ce type de dépendance est une anti-dépendance (WAR) :
- $S_1$  lit une valeur avant qu'elle ne soit écrasée par  $S_2$ .

```
for (int i = 0; i < N; i++) {  
    for (int j = 1; j < N; j++) {  
        A[i][j] = A[i][j-1] + 1; // S1  
    }  
}
```

