

# Architectures parallèles

Xavier JUVIGNY

ONERA

Novembre 2020

# Plan du cours

- 1 Prérequis et finalité du cours
- 2 Introduction
  - Motivations
- 3 Architecture des ordinateurs parallèles
  - Architecture SIMD
  - Architecture MISD
  - Architecture MIMD
- 4 Mémoire partagée, mémoire distribuée et mémoire NUMA
  - Mémoire partagée
    - Mémoire entrelacée
    - Mémoire cache
  - Mémoire distribuée
  - Architecture NUMA

# Prérequis et finalité du cours

## Prérequis

- Une bonne expérience en programmation
- Une bonne maîtrise du C++

## Finalité du cours

- Connaître les concepts fondamentaux de la programmation parallèle (hardware et software)
- Maîtriser les outils de mesure de performance adaptés à la programmation parallèle
- Être capable de concevoir un programme parallèle relativement complexe
- Notion de programmation parallèle distribué
- Notion de programmation parallèle partagé
- Notion de programmation parallèle GPGPU

# Loi de Gordon Moore : la fin en 2017 ?

## Loi de Moore

Nombre de transistors double dans les processeurs tous les 24 mois.

## Loi de Moore et ses limitations

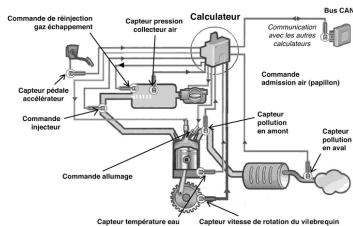
- Loi auto-réalisatrice énoncé en 1965
- Sert de feuille de route aux industriels
- Hausse des fréquences & diminution de la taille des circuits :  
**problème de chaleur à dissiper**
- Miniaturisation circuits : "mur quantique"  $\Rightarrow$  effet tunnel, ...

## Les alternatives

- **Architecture multi-cœur** : palie le problème de chaleur mais programmation plus complexe.
- Alternative au silicium ? Informatique quantique, processeurs neuromorphiques, processeurs 3D, transistors au graphène ?

# Motivation : Contrôle-commande

## Contrôle-commande sur une voiture



## Caractéristiques

Nombreux calculateurs dédiés à des fonctions diverses : freinage ABS, gestion moteur, éclairage, climatisation, etc.

Chaque calculateur doit optimiser un grand nombre de paramètre. Moteur à combustion : pression air, température, mélange, allumage, etc.

Le tout en temps réel et chaque calculateur est inter-dépendant avec les autres.

# Motivation : Contrôle-commande (suite)

## Contrôle commande centrale nucléaire

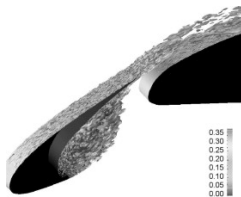
- algorithmes contraints en temps réel
- Nombreux paramètres et calculs complexes
- Un seul cœur de calcul peut ne pas suffire à répondre à la contrainte temps réel
- Exploitation de l'architecture multi-cœur : exécution concurrente de tâches **indépendantes**

## Exemple logiciel distribué : **labview**

Le passage à la parallélisation de tâche a permis un traitement des données quinze fois plus rapide pour **labview** sur certaines plateformes.

# Motivation : Simulation de phénomènes physiques

## Problème aéro-acoustique



## But

calculer le bruit généré par de petites turbulences au niveau du bec de sécurité d'une aile d'avion

- Petites turbulences  $\Rightarrow$  modèle précis & maillage très fin;
- **Sept milliards de sommets** avec 5 inconnues par sommets
- Mémoire nécessaire : **7 To**
- En séquentiel : **23 jours** pour simuler  $\frac{1}{100}^e$  de secondes

# Motivation : Deep Learning

**Deep Learning** : apprentissage de modèles de données, Modélise un réseau neuronal.

- internet ( reconnaissance d'images, traduction automatique, etc.)
- médecine ( détection cellules cancéreuses, drogues, etc.),
- sécurité et la défense ( détection faciale, surveillance vidéo, etc.)
- autonomie des machines (détection piéton, panneaux de signalisation )

Temps d'apprentissage :

- Ordinateur séquentiel, apprentissage  $> 1$  an
- GPGPU, en parallèle, apprentissage  $\approx 1$  mois.

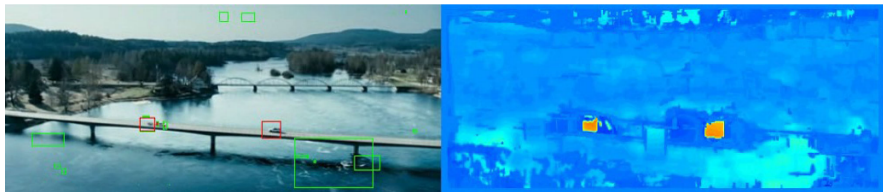
Mars 2016, Alphago bat champion du monde de GO (apprentissage supervisé).

Octobre 2017, Alphago zero bat Alphago 100 parties à zéro (deep learning)



# Motivation : Traitement de l'image

capteurs optiques pour navigation véhicules autonomes, super résolution sur images vidéos, etc.



GPGPU et parallélisme permettent un traitement d'un flux vidéo à trente images par secondes pour une résolution de  $1920 \times 1080$  pixels en temps réel.

# Taxonomie de Flynn

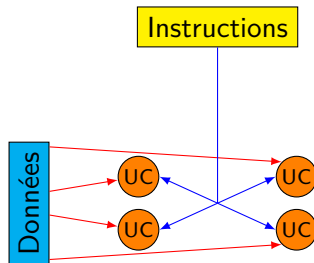
## Taxonomie de Flynn

Classification faite par Michael Flynn en 1967

- **SISD** : Simple Instruction, Simple Data : architecture classique, programmation séquentielle classique
- **SIMD** : Simple Instruction, Multiple Data : Une seule instruction exécutée à la fois mais appliquée sur plusieurs données simultanément : GPGPU, unités AVX et SSE.
- **MISD** : Multiple Instruction, Simple Data : plusieurs instructions exécutées traitant simultanément une seule donnée : architectures pipelines associées aux registres vectoriels.
- **MIMD** : Multiple Instruction, Multiple Data : plusieurs instructions qui traitent chacune des données différentes : ordinateurs multi-cœurs, calculateurs à mémoires distribuées.

De nos jours, un ordinateur possède plusieurs architectures parallèles.

# Architecture SIMD



Une unique instruction est exécutée simultanément par tous les unités de calcul sur ce type de machine, sur des données différentes.

# Architecture SIMD (suite)

- Mise en œuvre des algorithmes délicat.
- Instructions demandant sauts conditionnels très pénalisantes
- On utilise une technique de masque pour les sauts conditionnels.

Pour traiter les branches conditionnelles :

Code d'origine

Code compilé pour la machine

```
if ( a[i] >= 0 )  
    b[i] = c[i];  
else  
    b[i] = -d[i];
```

```
msk[i] = (a[i] >= 0); // = 1 si vrai, 0 sinon  
b[i] = msk[i] * c[i] - (1-msk[i]) * d[i];
```

# Les boucles conditionnelles en SIMD

## Exemple : boucle sur suites de Mandelbrot

- On calcule  $N$  suites  $z[i]$  avec  $z_{n+1}[i] = z_n^2[i] + c[i]$  avec  $c[i]$  complexe de module inférieur à deux.
- Pour chaque suite, converge ? diverge ?
- Diverge si  $|z_n[i]| \geq 2$ . On arrête de calculer la suite
- Arrêt au bout de `nIterMax`

## Code classique pour calculer la $i^{\text{e}}$ suite de Mandelbrot

```
const long nIterMax = 65000;
z[i] = 0;
iter[i] = 0;
while ( (abs (z[i] ) < 2) && (iter[i] < nIterMax) )
{
    z[i] = z[i]*z[i]+c[i];
    iter[i] += 1;
}
```

## Les boucles conditionnelles en SIMD (suite)

- Chaque unité de calcul doit itérer tant qu'une des suites n'a pas divergé;
- Utilisation système de masque pour que suites divergentes stagnent;
- On suppose unité dotée d'une instruction de réduction permettant de savoir si un des masques est vrai : `reduce(msk, op::or)` effectue un ou logique sur toutes les valeurs de `msk`.

### Code transformé

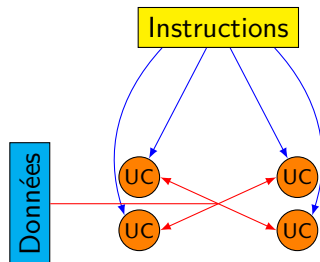
```
const long nIterMax = 65000;
z[i] = 0;
iter[i] = 0;
msk[i] = true;
while reduce(msk, op::or)
{ // Tant qu'un des masques d'une UC est vrai
  z[i] = msk[i]*(z[i]*z[i]+c[i]) + (1-msk[i])*z[i];
  iter[i] += msk[i];
  msk[i] = (abs(z[i]) < 2) && (iter[i] < nIterMax);
}
```

# Bilan des sauts conditionnels sur SIMD

## En résumé

- Les sauts conditionnels en SIMD sont traduits à l'aide d'une gestion de masque;
- Cependant cela à un coût important puisqu'une unité de calcul SIMD doit toujours exécuter les deux branchements et non l'un ou l'autre;
- La longueur des instructions dans chaque branche doit être limitée.
- Il faut donc éviter au possible les sauts conditionnels !

# Architecture MISD



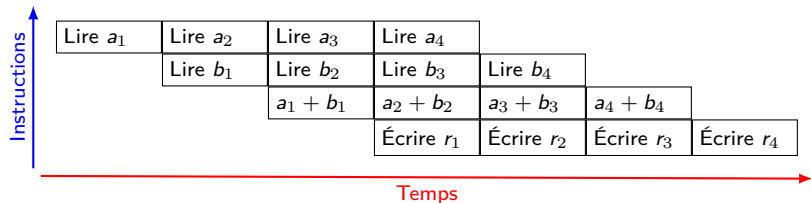
Plusieurs unités de calcul vont effectuer des opérations sur le même registre ( en principe, un registre vectoriel ), mais sur des parties différentes du registre.



## Architecture MISD (suite)

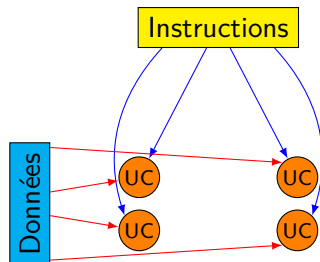
Ce type d'architecture recouvre actuellement les architectures pipelines associées à des registres vectoriels ( SSE – AVX ) sur les processeurs modernes :

```
for ( int i = 1; i <= 4; ++i )
    r[i] = a[i] + b[i];
```



On remplit les registres vectoriels en même temps qu'on commence l'addition sur les parties des registres de  $a$  et  $b$  déjà remplies.

# Architecture MIMD

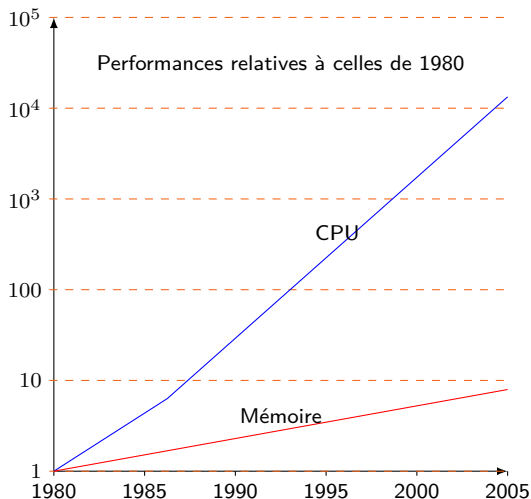


Cette architecture couvre une large gamme de machines, et est la plus commune actuellement :

- Les PCs actuels avec leurs multi-cœurs;
- Les calculateurs à mémoire distribuée ou NUMA;
- LES GPGPUs et leurs différentes unités de calculs.

# Performance relative de la mémoire vive

Composant de l'ordinateur limitant actuellement la vitesse de traitement des données :



# La mémoire partagée

Les unités de calcul partagent la même mémoire centrale au travers d'un bus mémoire.

La vitesse d'accès à la mémoire vive devenue critique pour les codes actuels.

Comment accélérer l'accès aux données ?

Principalement deux techniques complémentaires :

- La mémoire vive entrelacée;
- Une mémoire cache hiérarchique;

# La mémoire vive entrelacée

- Nombre de voies : Nombre de mémoire qui sont entrelacées;
- Largeur des voies : Nombre d'octets entre deux voies successives;

Exemple d'une mémoire entrelacée à quatre voies de largeur 3.

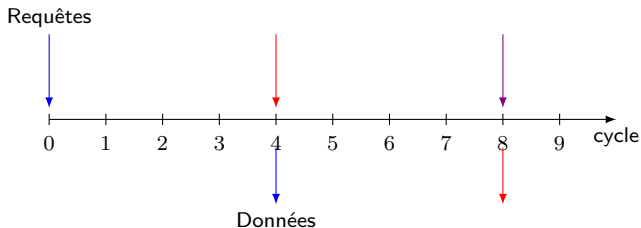


Un carré = un octet

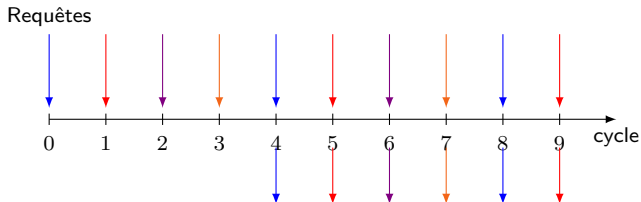
Le bloc  $i, i \in [0, n - 1]$  contient toutes les cellules dont les adresses sont égales à  $w \times (n \times a + i) + k$  avec  $k \in [0, w - 1]$  et  $a \in \mathbb{N}$ .

# La mémoire vive entrelacée (suite)

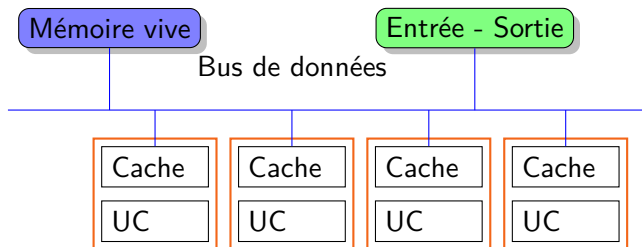
## Accès à une mémoire non entrelacée



## Accès à une mémoire entrelacée à quatre voies



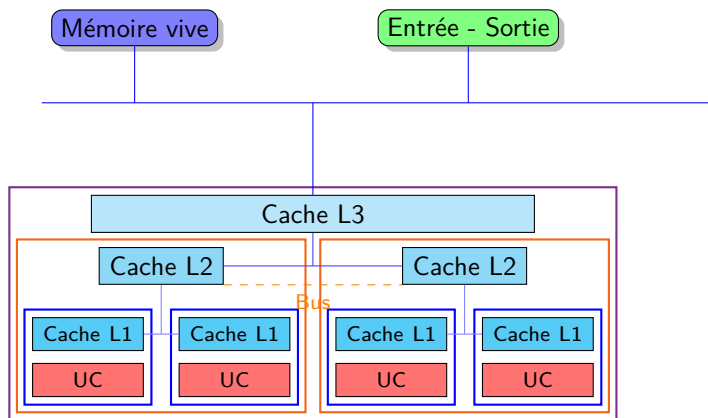
# Organisation de la mémoire cache sur une machine multi-processeurs



## Cohérence des données entre mémoires caches :

- Une seule mémoire cache contient la donnée : donnée valide, aucune synchronisation nécessaire;
- Donnée partagée avec d'autres caches : vérifier à chaque accès si modifiée par d'autres processeurs et alors la marquer comme invalide.
- Valeur modifiée dans le cache et valeur en mémoire vive plus à jour : Mettre à jour la mémoire vive si d'autres processeurs veulent lire la valeur;
- Valeur dans la mémoire cache invalide. La prochaine lecture de cette valeur déclenchera une lecture en mémoire vive.

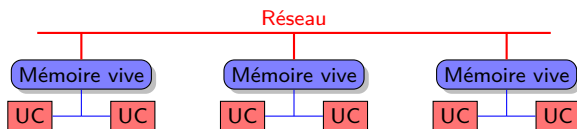
# Organisation de la mémoire cache sur une machine multi-cœur



Même problème de cohérence des caches, mais également entre les niveaux de cache ! Complexité croissante avec le nombre de niveau hiérarchique de cache !

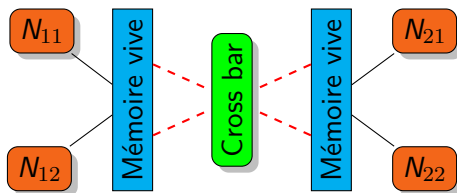


# Principe de la mémoire distribuée



- Chaque unité de calcul possède sa propre mémoire vive : l'ensemble unité de calcul + mémoire vive est appelée **nœud de calcul**;
- Les données sont échangées entre les nœuds de calcul au travers soit d'un bus spécialisé soit au travers un réseau intranet dédié;
- Sur réseau, c'est à la charge du programmeur de transiter les données d'un nœud à l'autre;
- Demande une programmation et des algorithmes spécifiques pour exploiter au mieux de type de machine;
- Permet d'exploiter plusieurs centaines de milliers de nœuds de calcul;
- Cependant limité par la consommation électrique !

# Architecture NUMA



- Machine contenant des nœuds de calcul;
- Chaque nœud contient plusieurs unités de calcul;
- Les nœuds de calcul sont reliés par un crossbar;
- Du point de vue programmeur, la mémoire est partagée;
- Mais les temps d'accès aux données varient fortement selon qu'une unité de calcul accède à une donnée se trouvant sur son nœud de calcul ou sur un nœud de calcul distant.

# Architecture NUMA

- On trouve des architectures NUMA dans les Xeons des processeurs INTEL. Chaque chipset est un nœud de calcul relié par un crossbar appelé **QPI** ( Quick Path Interconnect ) dans la terminologie d'INTEL;
- Programme classique multi-thread fonctionne mais peu optimiser;
- L'optimisation demande de fixer chaque thread à une unité de calcul : c'est **l'affinité**;
- Le programmeur peut ensuite contrôler la répartition des données sur les nœuds de calcul à l'aide d'une politique choisie parmi les suivantes ( non exhaustive ) :

**default** Allocation sur le nœud local où s'exécute le processus

**First Touch Policy** L'allocation ne se fait pour une page mémoire que lors du premier accès à une donnée de cette page. L'allocation se fait alors pour cette page sur le nœud local où s'exécute le processus

**Interleaving** L'allocation d'un tableau se fait en distribuant de façon cyclique les pages mémoires sur les différents nœuds de calcul