Fantastic Architectures and How to Use Them

How to Try, At Least

Pierre Guillou

Fontainebleau, 10 mai 2017

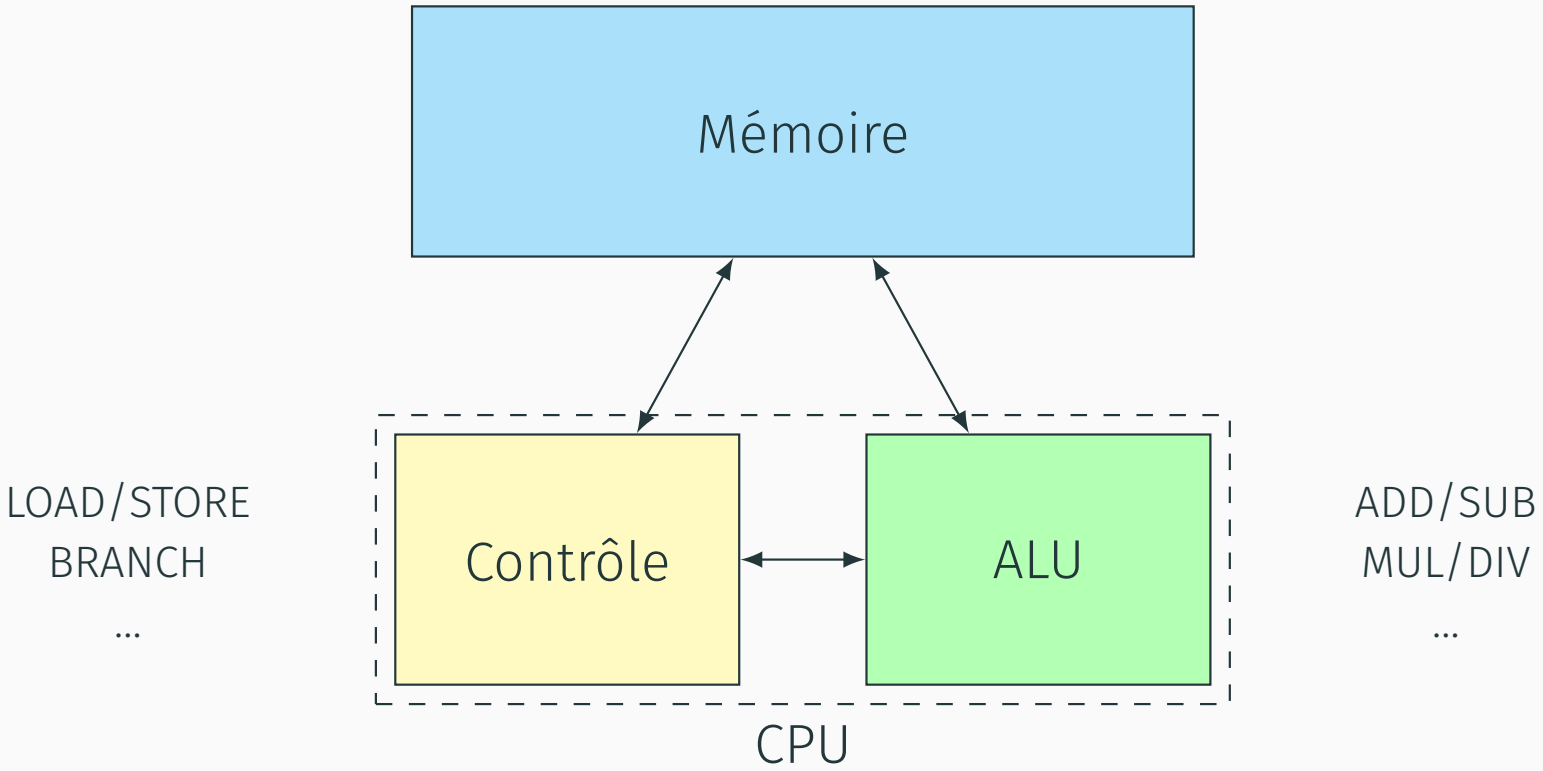
Machine Learning Club
MINES ParisTech, PSL Research University Paris


ONE DOES NOT SIMPLY
GET GOOD PERFORMANCE IN PYTHON
imgflip.com

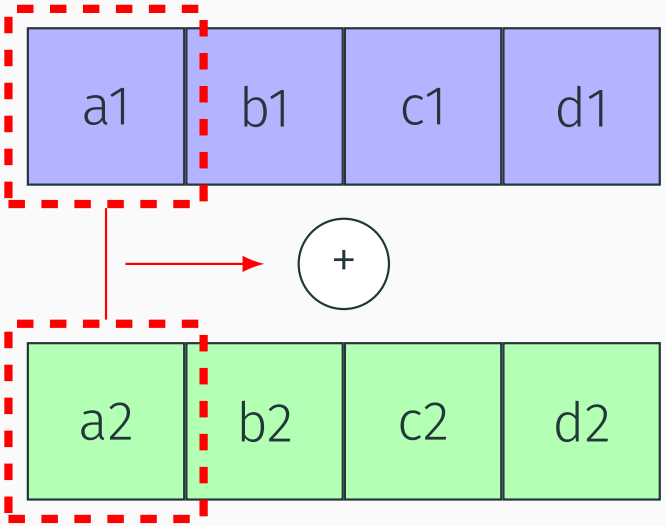
Contenu

Plein d’architectures matérielles

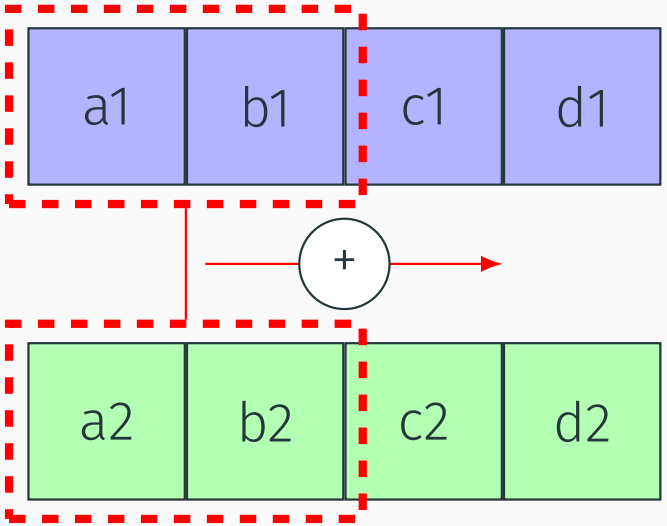
Modèle : architecture de von Neumann



Unités de calcul vectorielles



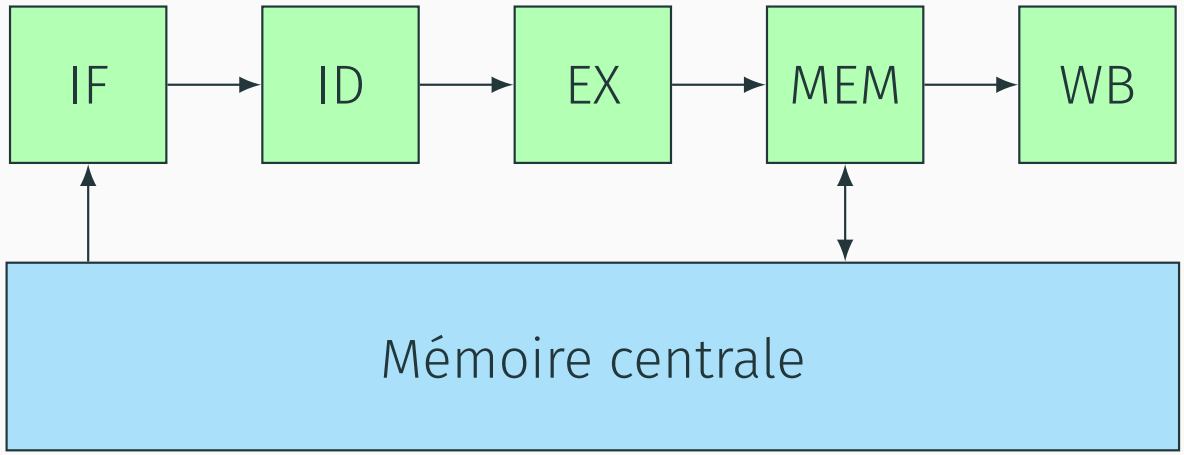
Sans vectorisation



Avec vectorisation

SSE (1999, 128 bits) $\longrightarrow$ AVX-512 (2013, 512 bits)

Parallélisme d'instructions



Exécution en pipeline instructions à la file

Prédiction de branchements accélérer le fetch

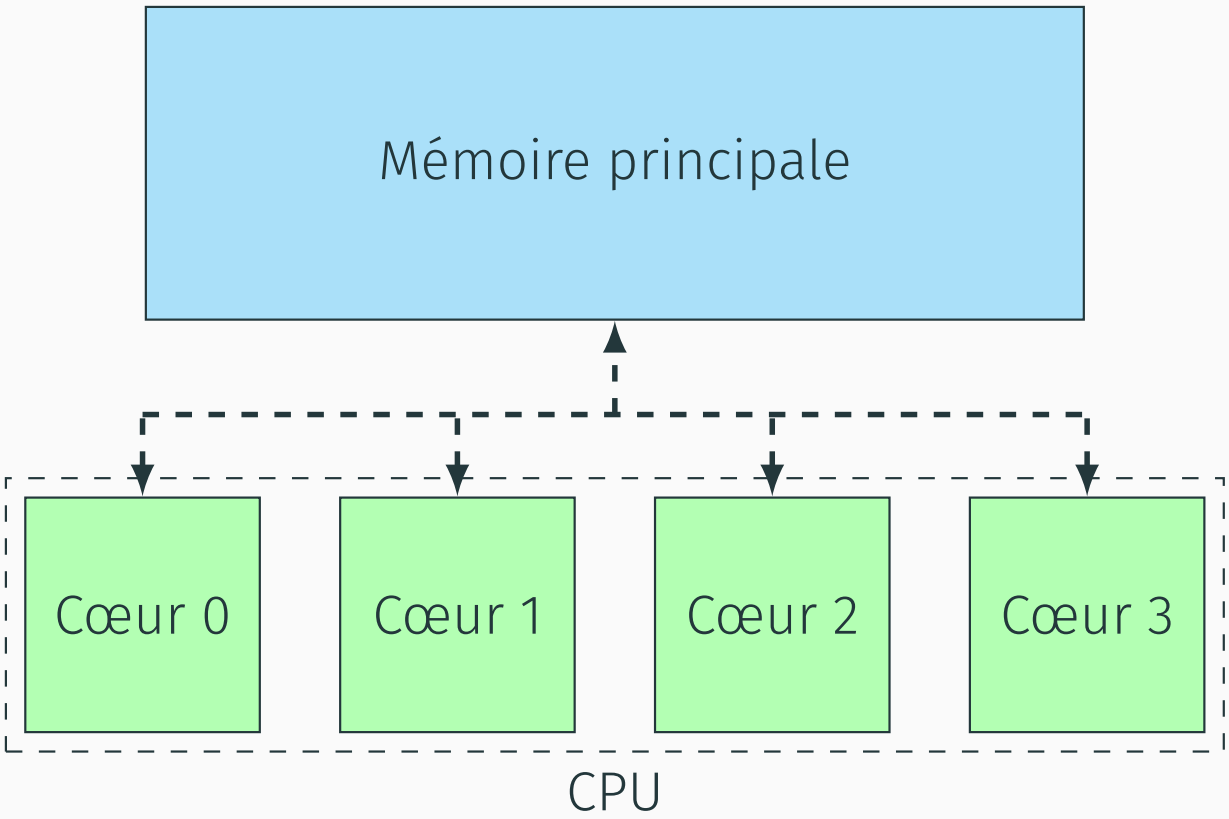
Exécution out-of-order réorganisation dynamique du pipeline

- déterminer les dépendances entre instructions

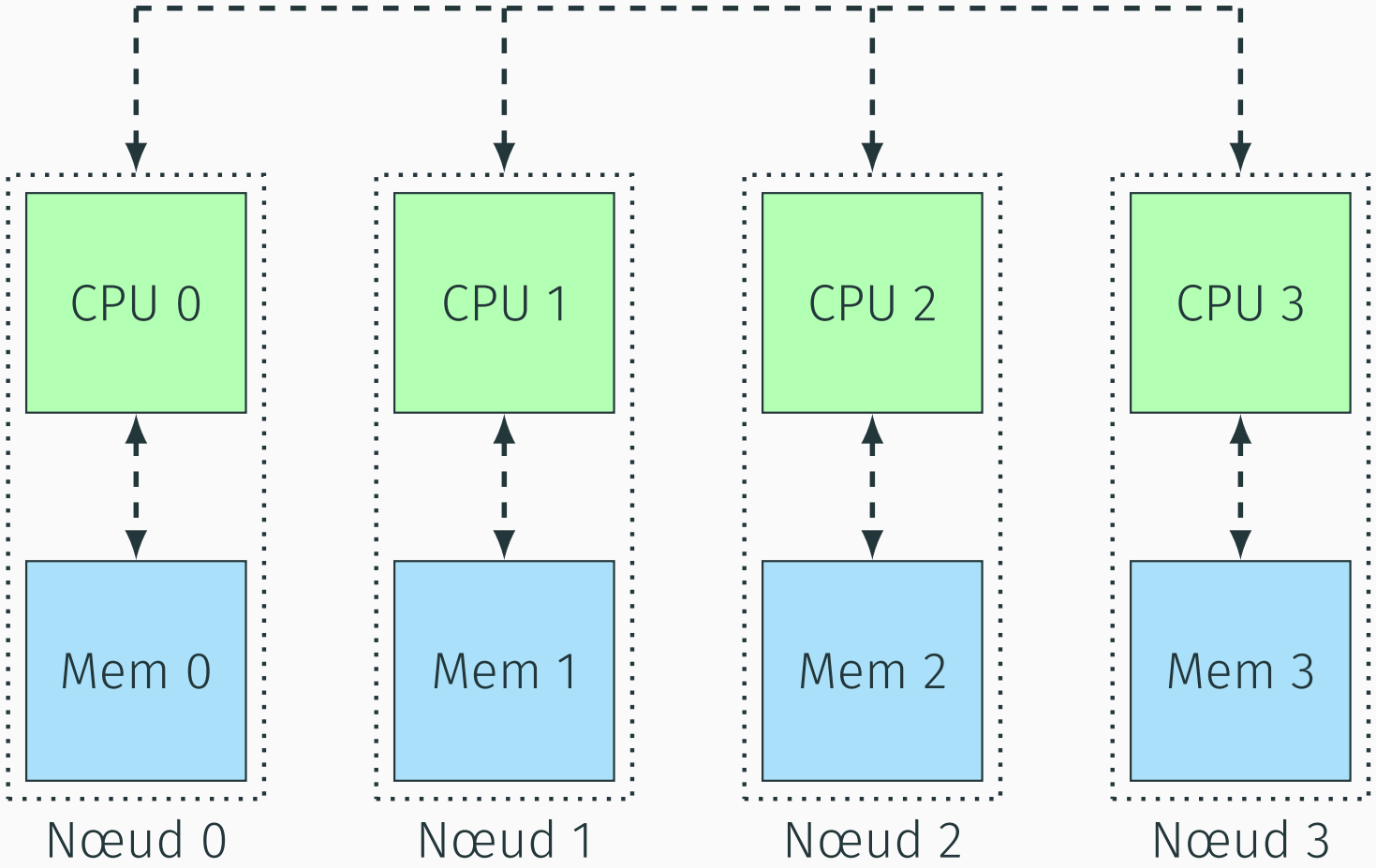
Very Long Instruction Word plusieurs instructions/cycle

- bundle d'instructions de différent type

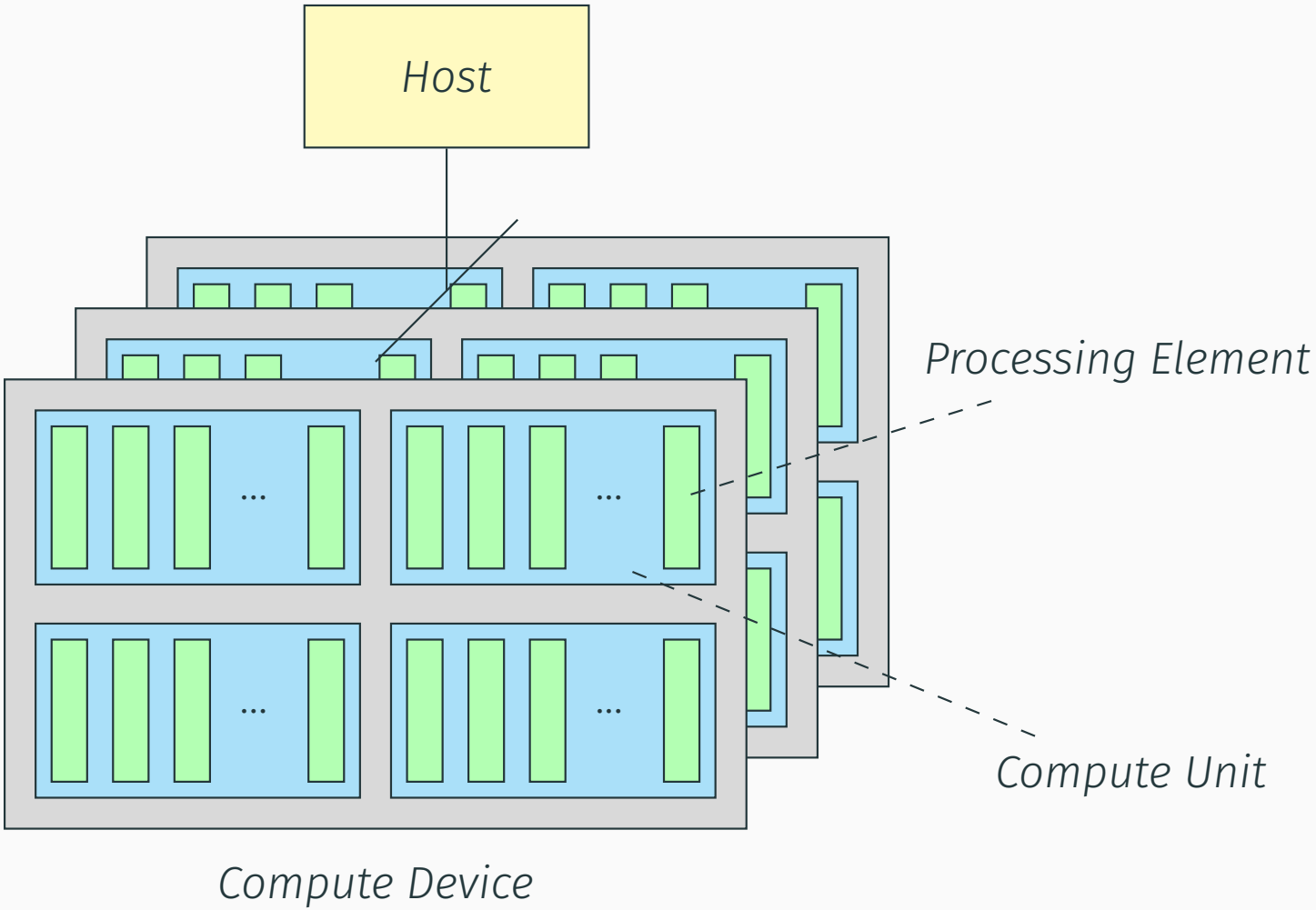
Architectures à mémoire partagée



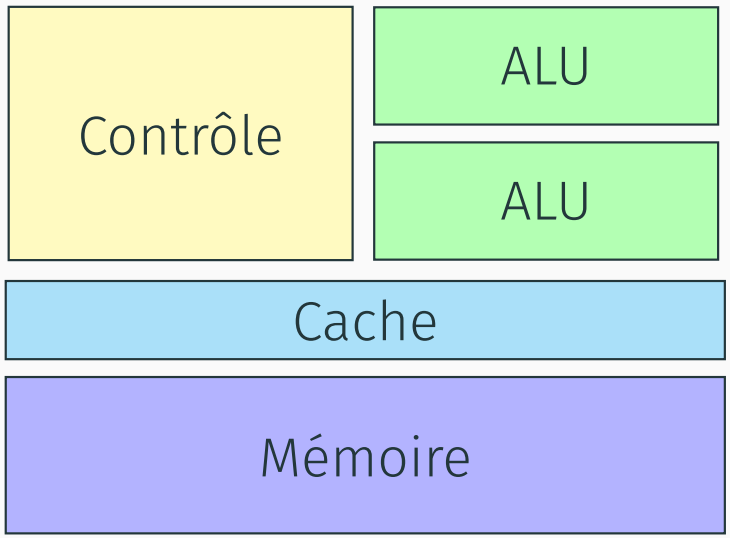
Architectures à mémoire distribuée/répartie



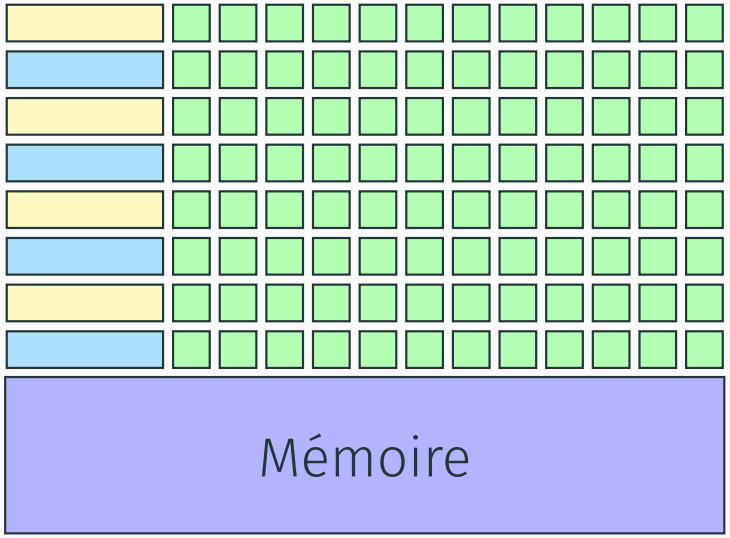
Architectures hétérogènes



CPU vs. GPU



Processeur central



Processeur graphique

Taxonomie de Flynn

	Single Data Stream	Multiple Data Streams
Single Instr. Stream	SISD	SIMD
Multiple Instr. Streams	MISD	MIMD

Take-off list

Répartition de la mémoire et nature des unités de calcul

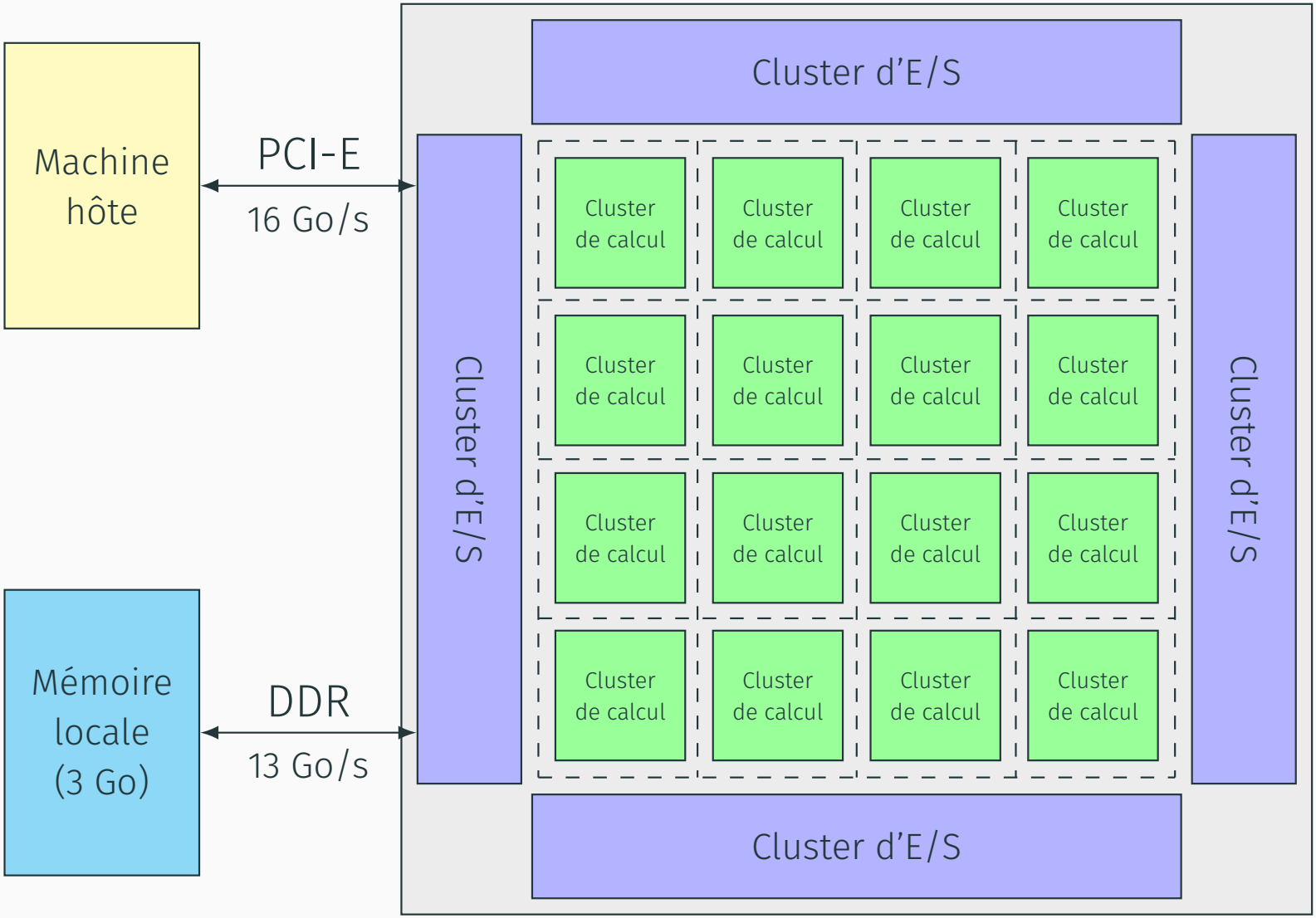
Mémoire partagée processeurs multi-cœurs modernes

Mémoire distribuée supercalculateurs

Architectures hétérogènes CPU + GPU

+ potentiel parallélisme à l'intérieur des unités de calcul

Un processeur *manycore* : le MPPA de Kalray



16 clusters × (16 cœurs VLIW/SIMD + 2 Mo)

Paralléliser c’est compliqué

Deux grandes classes de parallélisme

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

Parallélisme de données

```
sort(A, n, cmp_func);
fill(B, n, rand);
```

Parallélisme de tâches

Bugs de parallélisation

```
T1   a = 1;
T0   a = 3;
T1   a = a * 2;
```

Accès concurrents

```
T0   lock(a);
T1   lock(b);
     ...
T0   lock(b);
T1   lock(a);
```

Interblocages

$\longrightarrow$ problèmes de *synchronisation*

Limites de la parallélisation : loi d'Amdahl

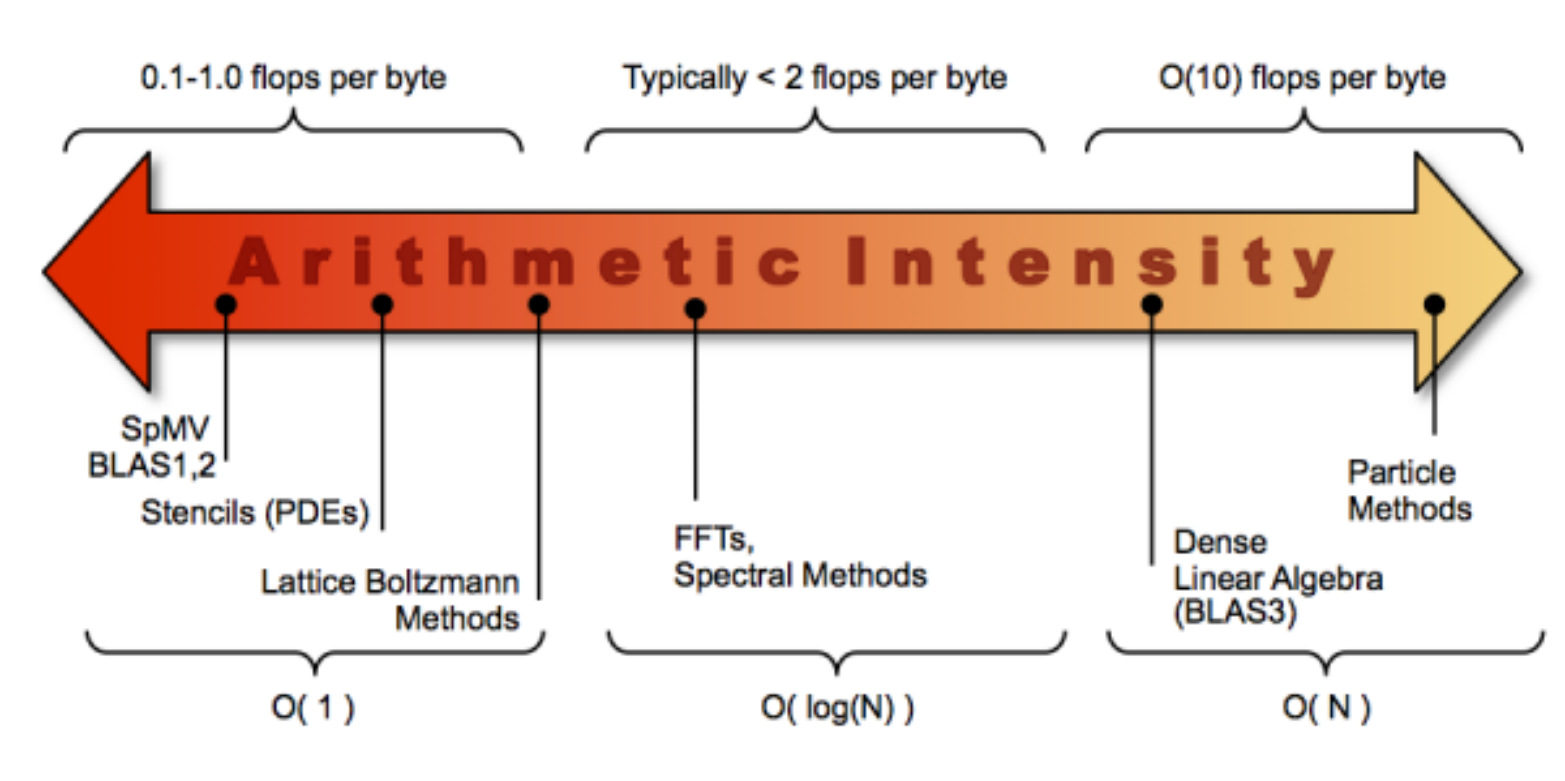
$$T = T_{par} + T_{seq} \tag{1}$$
$$= \alpha \times T + (1 - \alpha) \times T \tag{2}$$

$$Acc(p) = \frac{T_{par} + T_{seq}}{\frac{T_{par}}{p} + T_{seq}} \tag{3}$$
$$= \frac{1}{\frac{\alpha}{p} + (1 - \alpha)} \xrightarrow[p \to \infty]{} \frac{1}{1 - \alpha} \tag{4}$$

$\longrightarrow$ parallélisation limitée par portion séquentielle

Bande passante mémoire et intensité arithmétique

$$T_{exec} = T_{calcul} + T_{coms} \tag{5}$$



En pratique : des modèles de programmation parallèles

Tirer parti des architectures parallèles

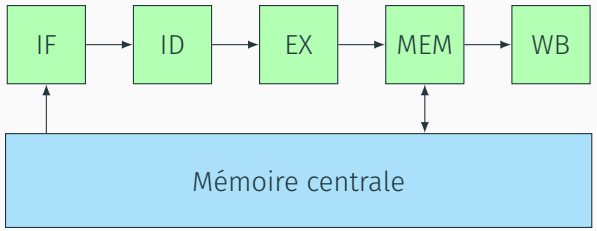
Modèles de programmation

- bibliothèques ou extensions de langages
- faciliter le support d'architectures parallèles

Enjeux : les *3P*

• programmabilité	temps de développement, maintenance
• portabilité	cibles matérielles
• performance	temps d'exécution, énergie

Parallélisme d'instruction



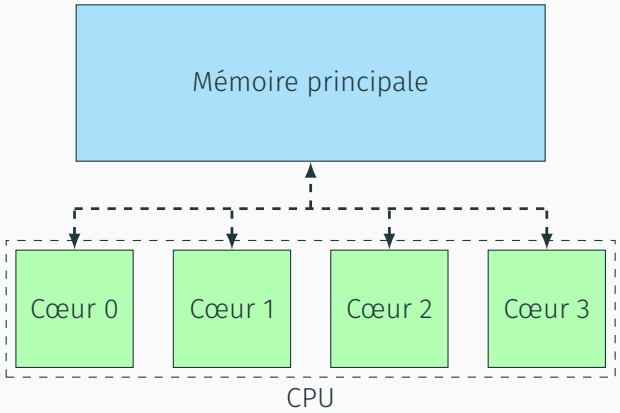
Géré par le compilateur ou les unités de calcul

Out-of-Order géré par le hardware

Vectorisation support compilateur ou via intrinsèques C

VLIW support compilateur ou assembleur à la main

Modèles pour mémoire partagée



Division du travail en *processus* ou *threads*

processus communication par des *tubes* (*pipes*)

threads accès directs à la mémoire $\longrightarrow$ synchronisation !

Modèles de programmation par threads

pthreads gestion explicite, standard POSIX

OpenMP directives préprocesseurs

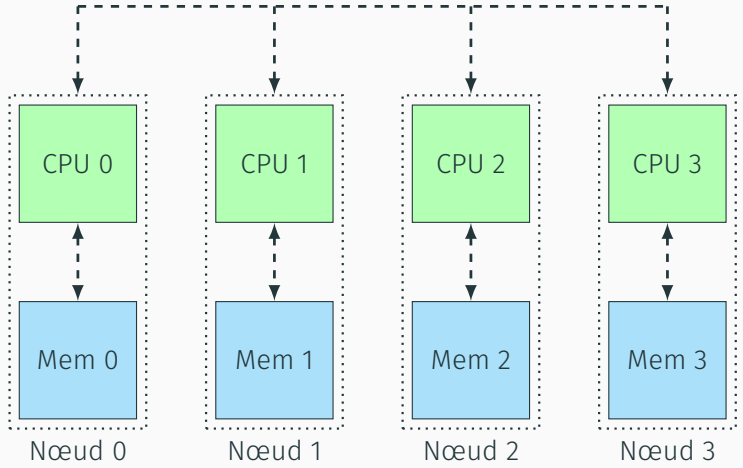
Primitives de synchronisation

mutex verrouille un emplacement mémoire

sémaphore limite l'accès simultané à une adresse

instruction atomique assure la cohérence des données

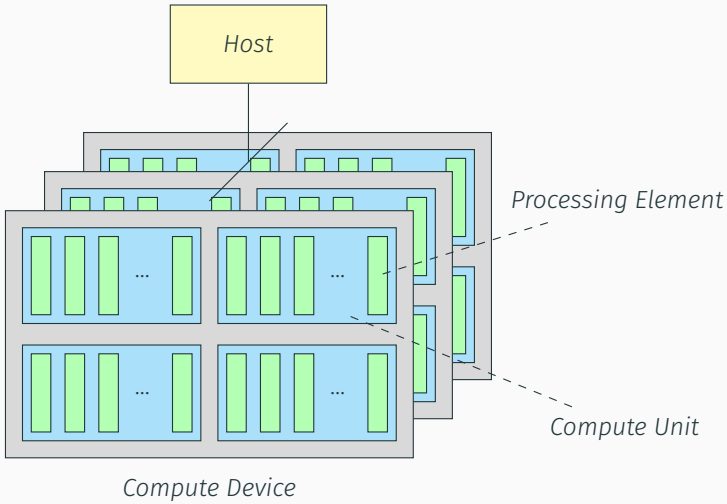
Modèles pour mémoire distribuée



Chaque nœud exécute un processus

- communications explicites — MPI
- mémoire unifiée — PGAS

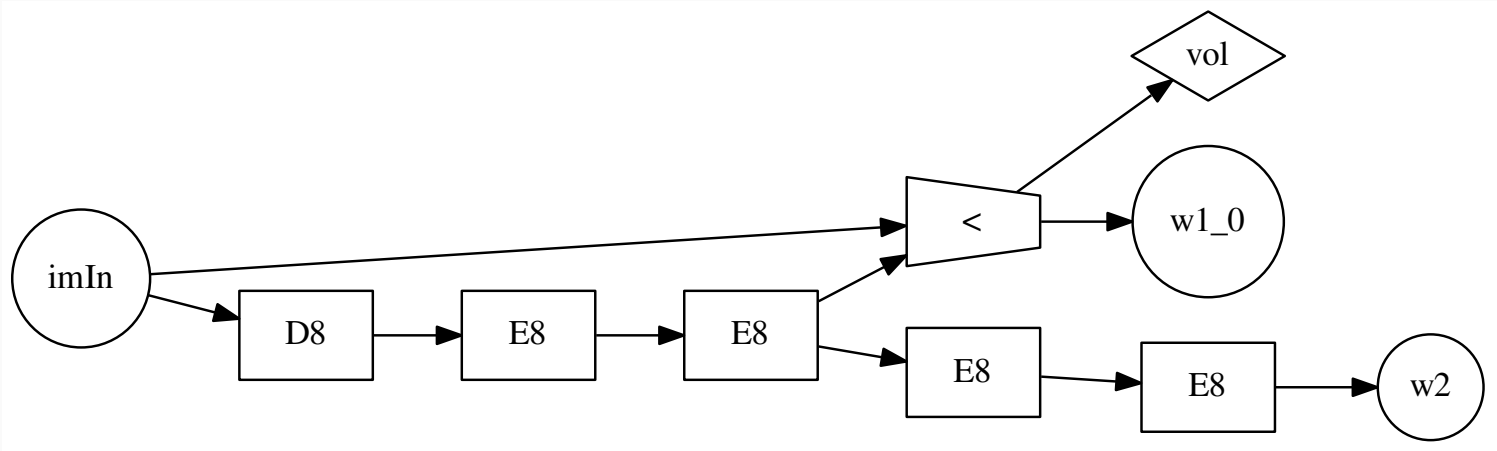
Modèles pour architectures hétérogènes



L'hôte orchestre, l'accélérateur exécute des *noyaux de calcul*

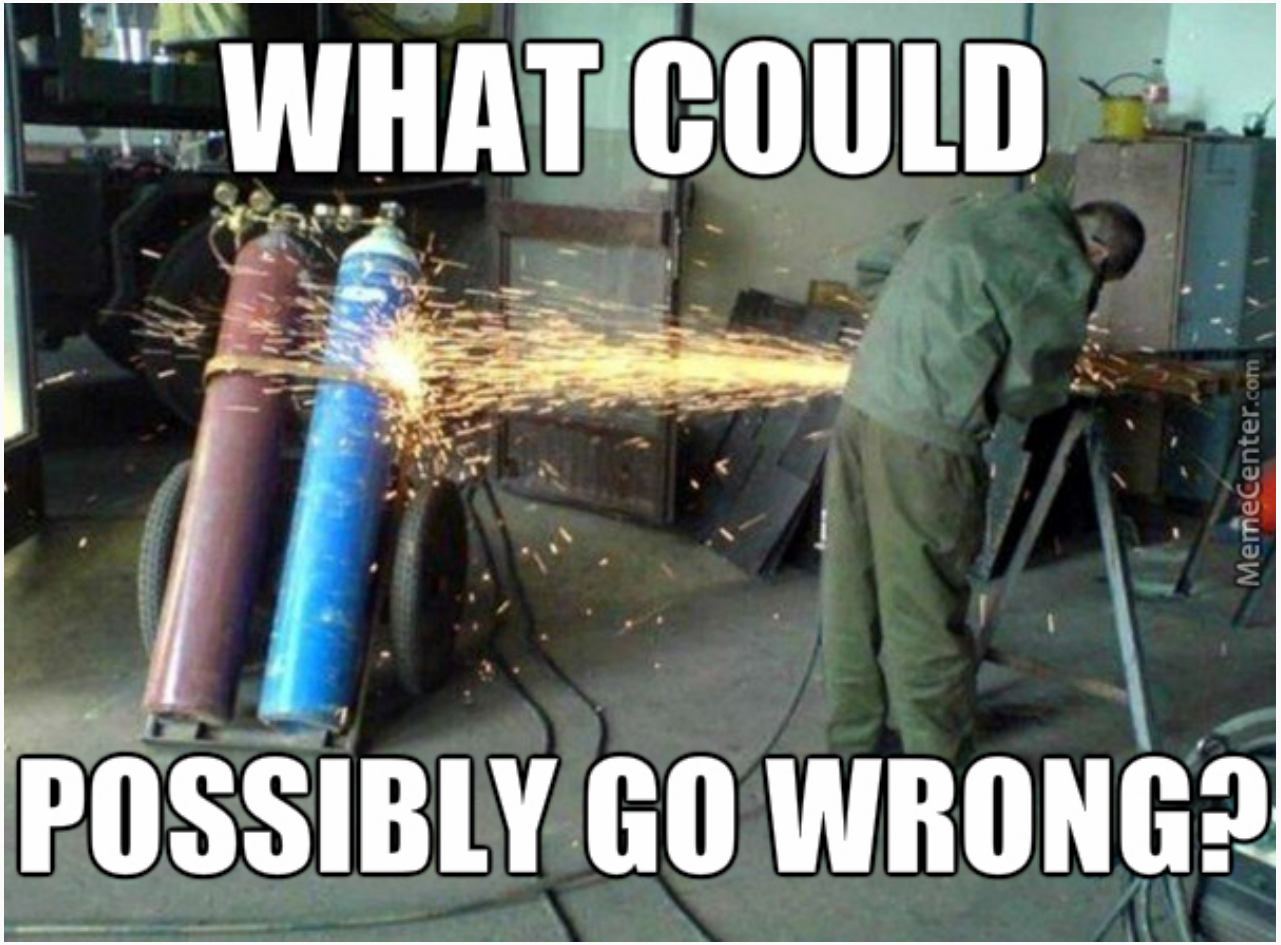
- CUDA — spécifique GPUs NVIDIA
- OpenCL — plus générique (GPUs + FPGAs)

Modèle flot de données

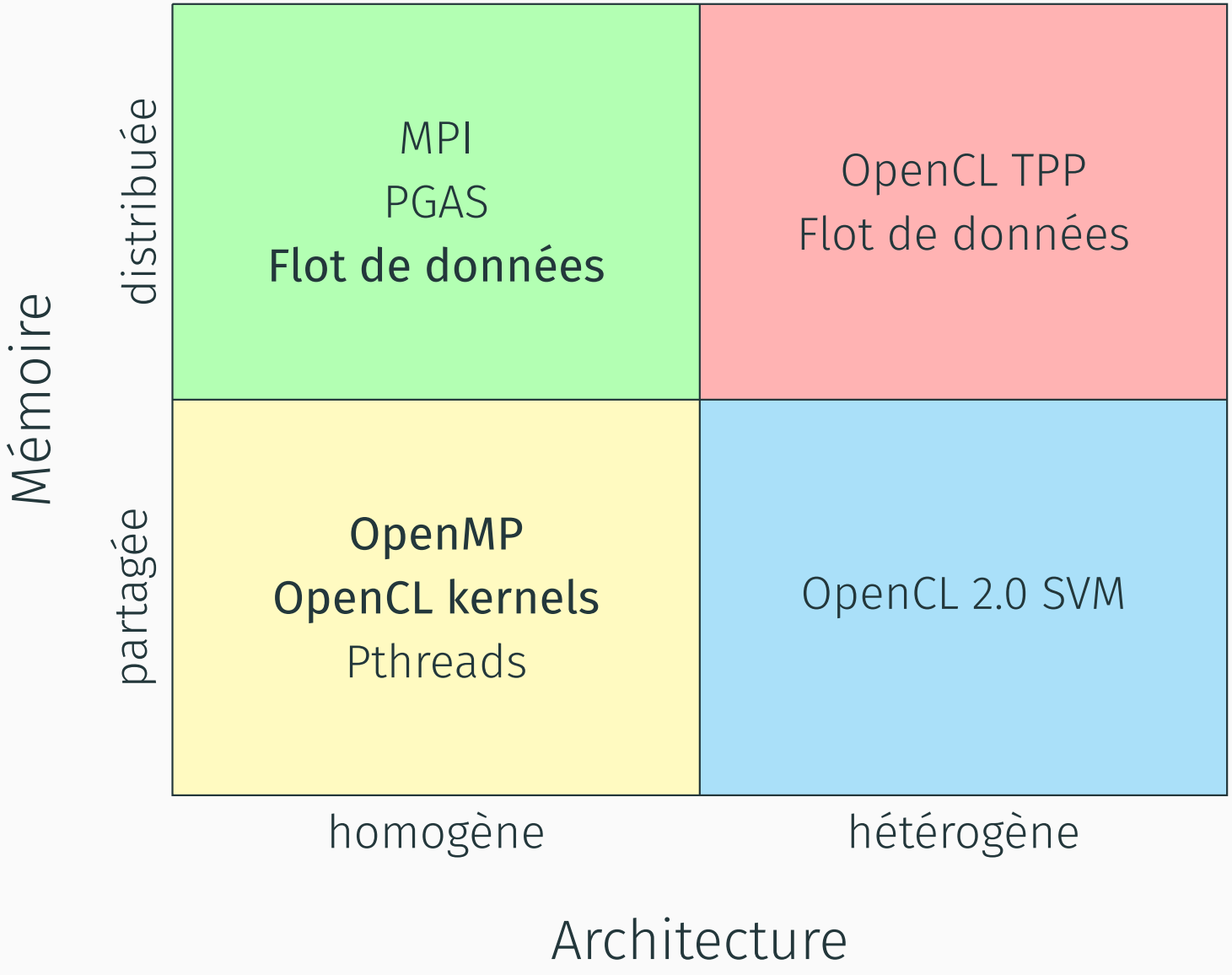


- abstraction de l'architecture
- applications = graphe de tâches
- parallélisme intrinsèque : tâches, pipeline, données
- optimisations : fusion, fission, placement
- adapté aux architectures massivement parallèles actuelles

Its Demo Tiem



Classes d'architectures et modèles de programmation



Vers des bibliothèques de haut niveau

Du Python sur mon GPU ??

C, la *lingua franca* pour programmer le hardware

- modèle mémoire simple — `structs`, `unions`
- contrôle direct sur le hardware — pointers, `registers`
- portabilité

Des *bindings* vers des bibliothèques C

1. bibliothèques C génériques — OpenMP, CUDA
2. bibliothèques spécifiques — Blas, CUDNN
3. bibliothèques interfaces de haut niveau — Numpy, Tensorflow

Une voie parallèle

Génération automatique de code

- interface haut niveau $\longrightarrow$ implémentation spécifique
- optimisations spécifiques au { domaine $\times$ hardware }
  - fusion/fission d'opérateurs
  - déport sur accélérateur graphique
  - communication des données

Pour le *deep learning*

- description d'un graphe de neurone en Python/...
- génération de code bas niveau spécifique OpenMP, CUDA, MPI

Conclusion

1. plein d'architectures parallèles compliquées
2. des modèles de programmation sont là pour nous aider
3. les compilateurs (et les Makefiles) peuvent améliorer les perfs
4. les vrais chercheurs peuvent coder en Python sur GPU

$\longrightarrow$ on a toujours besoin d'un bon technicien

Questions?

Fantastic Architectures and How to Use Them

How to Try, At Least

Pierre Guillou

Fontainebleau, 10 mai 2017

Machine Learning Club
MINES ParisTech, PSL Research University Paris