

# Introduction au calcul GP-GPU

## Partie 1 : Mise en contexte

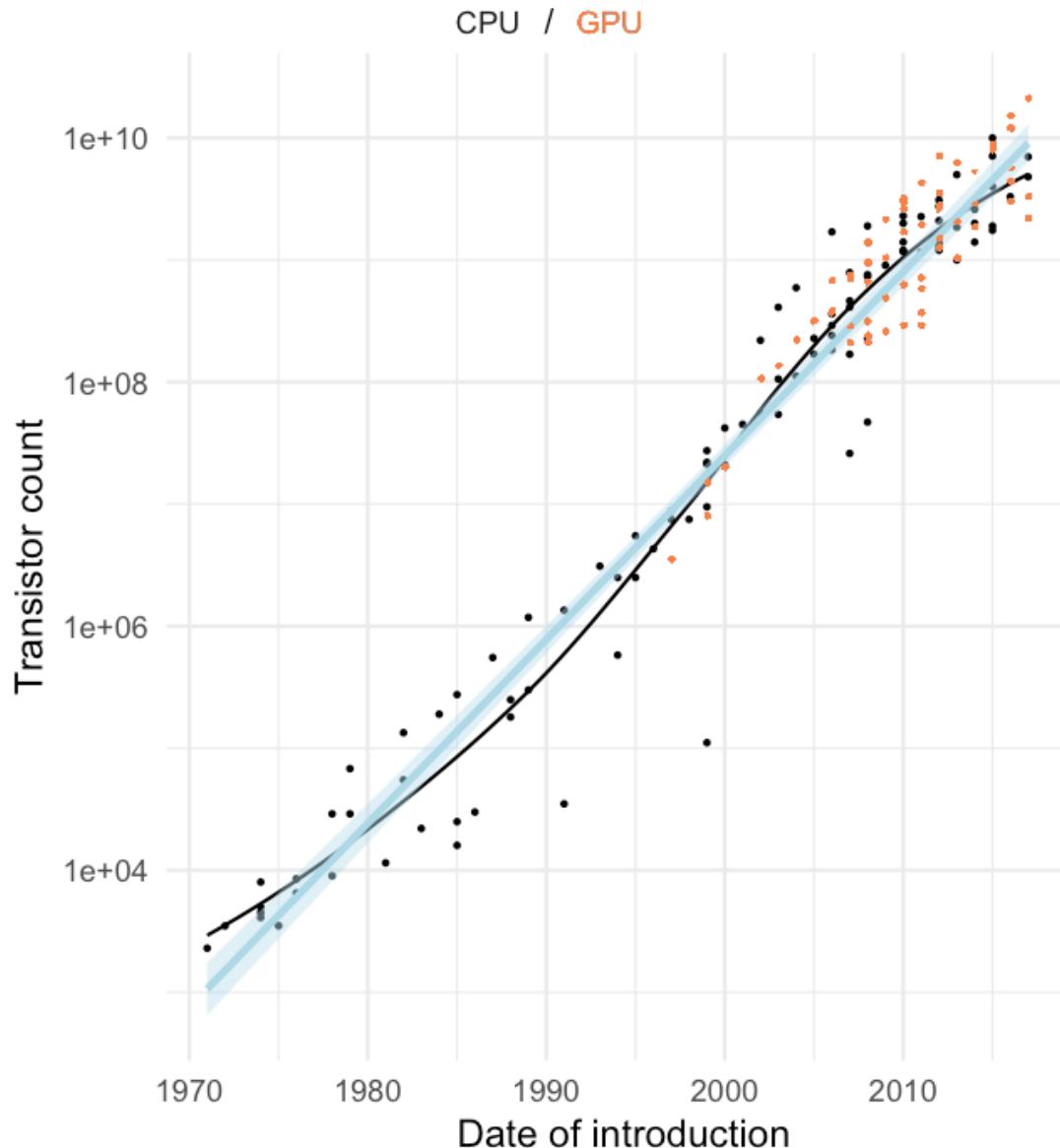
Laurent Risser

Institut de mathématiques de Toulouse  
[lrisser@math.univ-toulouse.fr](mailto:lrisser@math.univ-toulouse.fr)

# Préambule - loi de Moore

Moore's Law continued

Still linear



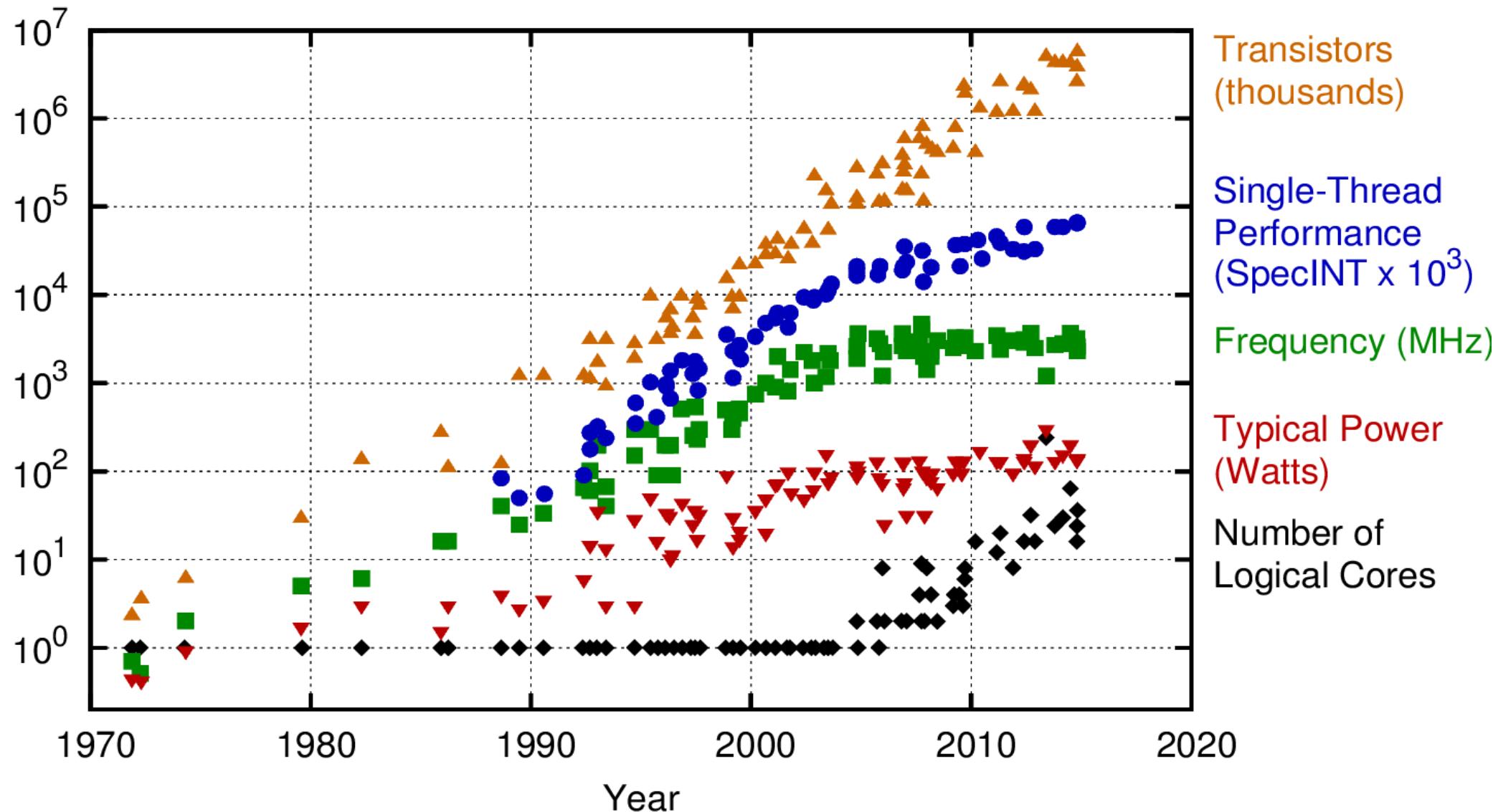
## Loi de Moore (fin des années 60)

Deux fois plus de transistors dans un circuit de même taille tous les 18 mois

## Déclaration de Gordon Moore en 1997

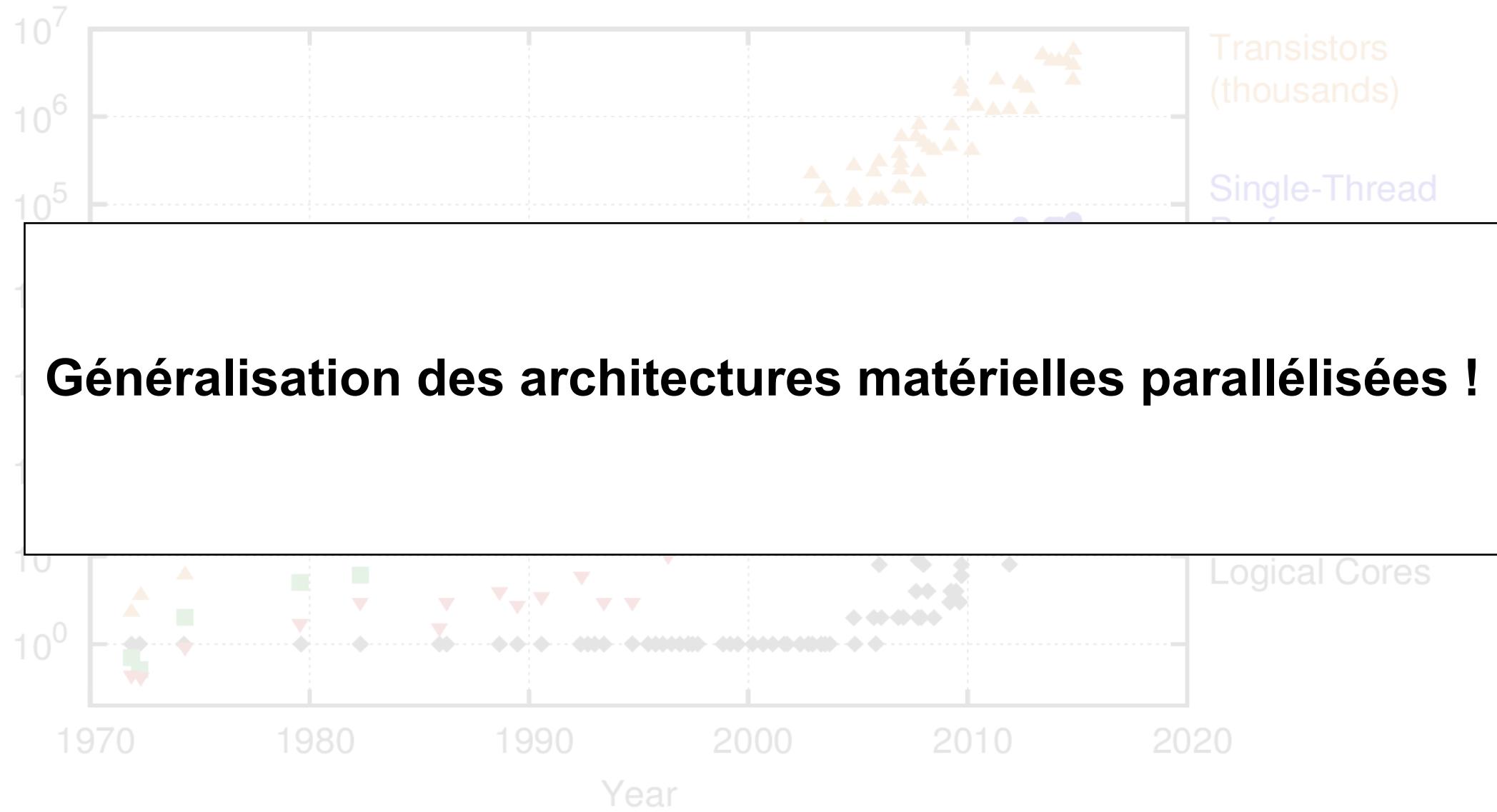
Fin de la loi en 2017 dû à une limite physique.

## 40 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
 New plot and data collected for 2010-2015 by K. Rupp

### 40 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

# Préambule - Emergence des GPU

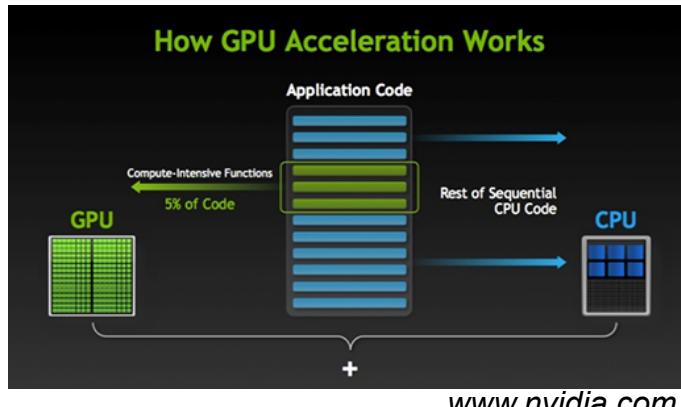
## Années 1990 :



Need for speed 2

- Fort développement des jeux vidéos 3D
- Émergence des cartes graphiques (GPU) dédiées à la 3D
- Cartes spécialisées dans la parallélisation de fonction d'algèbre linéaire (multiplications matrices/vecteurs).

## Années 2000 :



- 2001 : Ouverture aux programmateurs du pipeline graphique des cartes Nvidia.  
→ Développement du calcul GPGPU (General Purpose GPU).

- 2007 : Lancement du langage CUDA chez Nvidia  
→ Ouvre clairement la possibilité de largement paralléliser des calculs sur GPU.

2009 : Lancement d'OpenCL

- concurrent *libre* de CUDA qui fonctionne sur multiples plateformes (GPU Nvidia, ATI, AMD; CPU INTEL, ...; Windows, MacOS, Linux, Android, iOS)

## Ces dernières années :

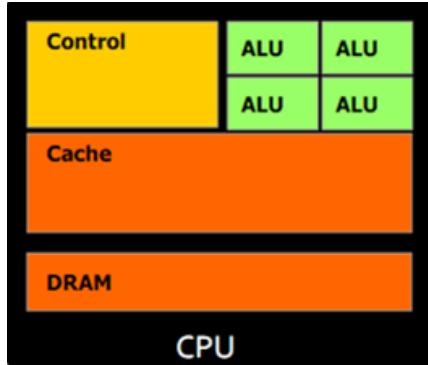
- Utilisation courante de CUDA et OpenCL en calcul intensif
- Succès du Deep Learning et de XGBoost fortement lié au calcul GPGPU

## **PREAMBULE**

**PARTIE 1 : Principes du calcul GPGPU**

**PARTIE 2 : En pratique**

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

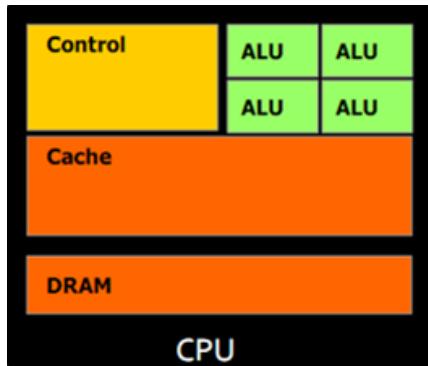
**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

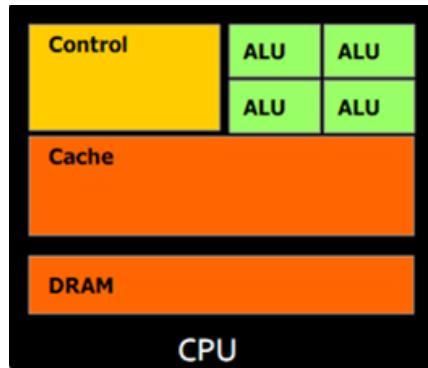
**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

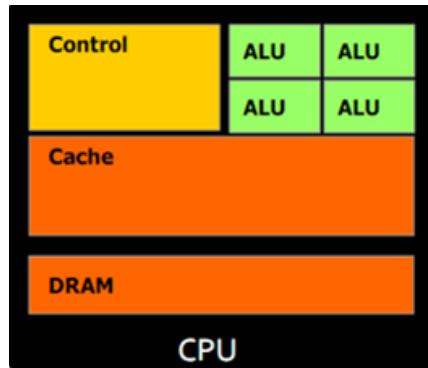
## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

Structuration possible dans la DRAM ou le Cache

$$1 \ 3 \ \dots \ -2 \quad 2 \ 1 \ \dots \ 0 \quad \dots \quad 0 \ 1 \ \dots \ 3 \quad 1 \ -1 \ 0 \ \dots \ 1 \quad ? \ ? \ ? \ \dots \ ?$$

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

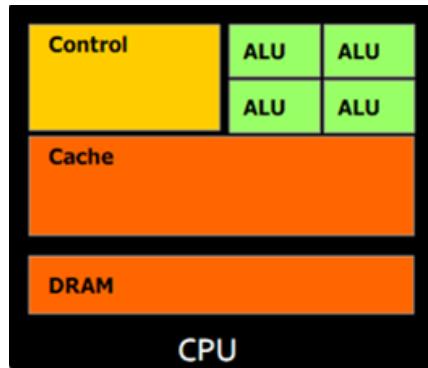
## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

Opérations + et \* sur différentes zones mémoires par le CPU

$$1 \boxed{3} \dots -2 \ 2 \ 1 \dots 0 \dots 0 \ 1 \dots 3 \boxed{1} -1 \ 0 \dots 1 \ \boxed{?} \ ? \ ? \dots ?$$

# 1) Calcul GPGPU – CPU



**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

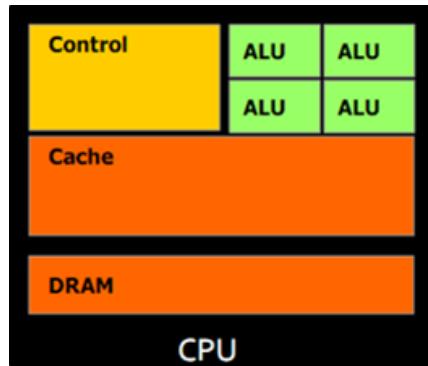
## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & \boxed{3} & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

Opérations + et \* sur différentes zones mémoires par le CPU

$$1 \boxed{3} \dots -2 2 1 \dots 0 \dots 0 1 \dots 3 1 \boxed{-1} 0 \dots 1 \boxed{?} ? ? \dots ?$$

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

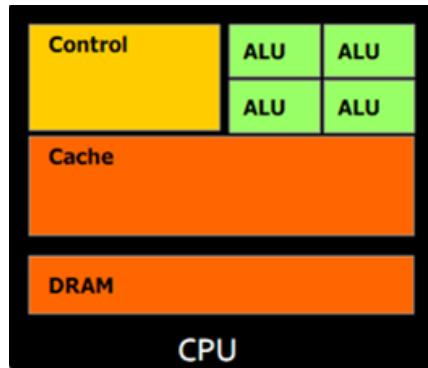
## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & 3 & \dots & \dots & 2 \\ 2 & 1 & \dots & \dots & 0 \\ -1 & -2 & \dots & \dots & 0 \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ 0 & 1 & \dots & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \\ ? \end{pmatrix}$$

Opérations + et \* sur différentes zones mémoires par le CPU

$$1 \ 3 \ \dots \boxed{-2} \ 2 \ 1 \ \dots \ 0 \ \dots \ 0 \ 1 \ \dots \ 3 \ 1 \ -1 \ 0 \ \dots \boxed{1} \ \boxed{?} \ ? \ ? \ ? \ \dots \ ?$$

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

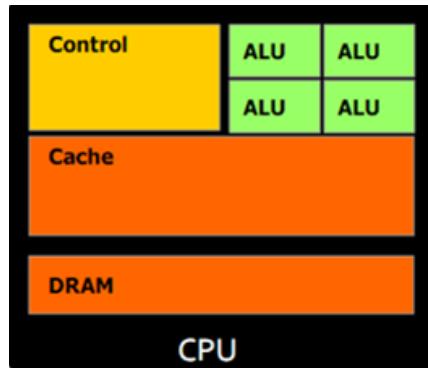
## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

Opérations + et \* sur différentes zones mémoires par le CPU

$$1 \ 3 \ \dots \ -2 \boxed{2} \ 1 \ \dots \ 0 \ \dots \ 0 \ 1 \ \dots \ 3 \boxed{1} \ -1 \ 0 \ \dots \ 1 \ \boxed{?} \ \boxed{?} \ \dots \ \boxed{?}$$

# 1) Calcul GPGPU – CPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

## Illustration : Multiplication matrice / vecteur

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \\ ? \end{pmatrix}$$

Opérations + et \* sur différentes zones mémoires par le CPU

$$1 \ 3 \ \dots \ -2 \ 2 \ 1 \ \dots \ 0 \ \dots \ 0 \ 1 \ \dots \ 3 \ 1 \ -1 \ 0 \ \dots \ 1 \ ? \ ? \ ? \ \dots \ ?$$

# 1) Calcul GPGPU – GPU

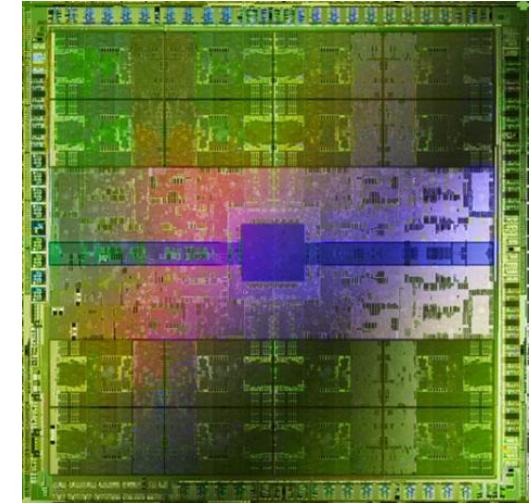
## Intuition naïve (et partiellement fausse)

Un GPU donne accès à des centaines de processeurs qui vont traiter de manière parallèle mes données en mémoire, d'où un gain évident en temps de calculs.

## Réalité matérielle

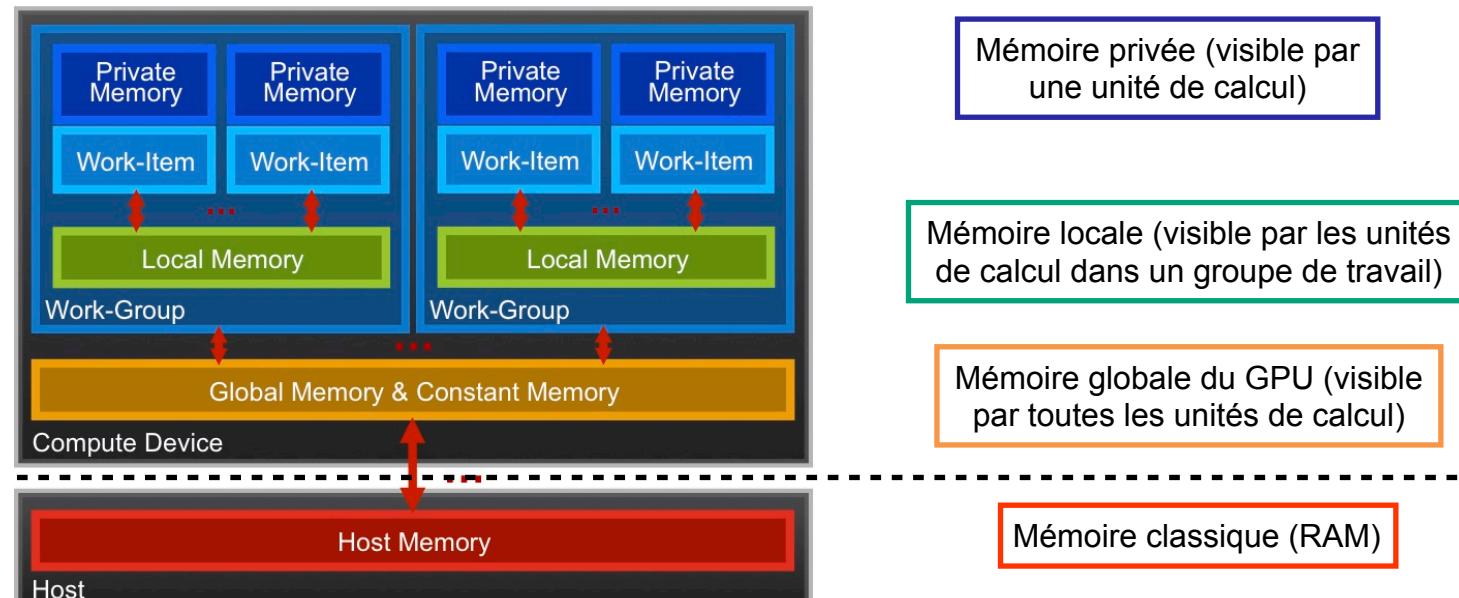
Modèle de mémoire complexe dû à :

- Contraintes physiques
- Coût des différents niveaux de mémoires (caches L1, L2 et L3)

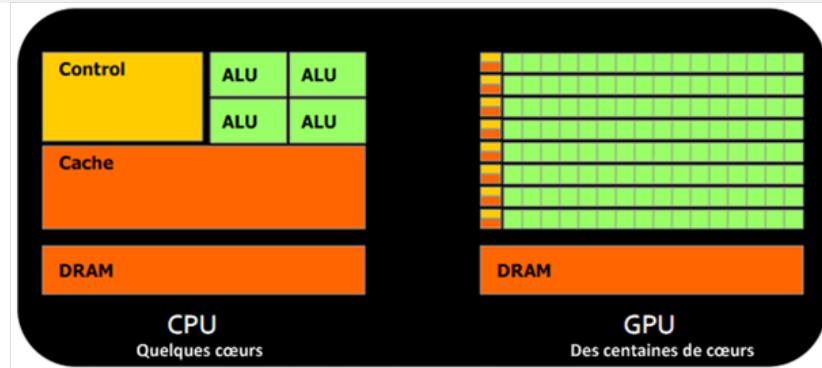


NVIDIA® Tesla® C2090

## Modèle de mémoire



# 1) Calcul GPGPU – GPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

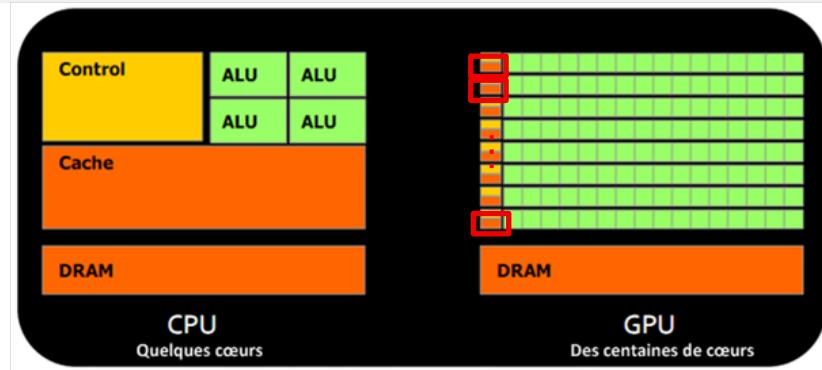
**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \\ ? \end{pmatrix} \xrightarrow{\text{DRAM}} 1\ 3\dots-2\ 2\ 1\dots0\dots0\ 1\dots3\ 1-1\ 0\dots1\ 1\dots1$$

# 1) Calcul GPGPU – GPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

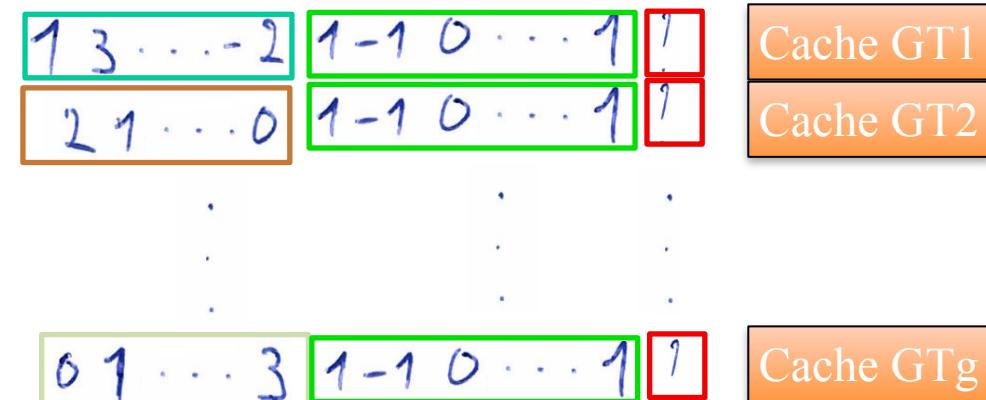
**Control** : Coordonne les ALU et la mémoire

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

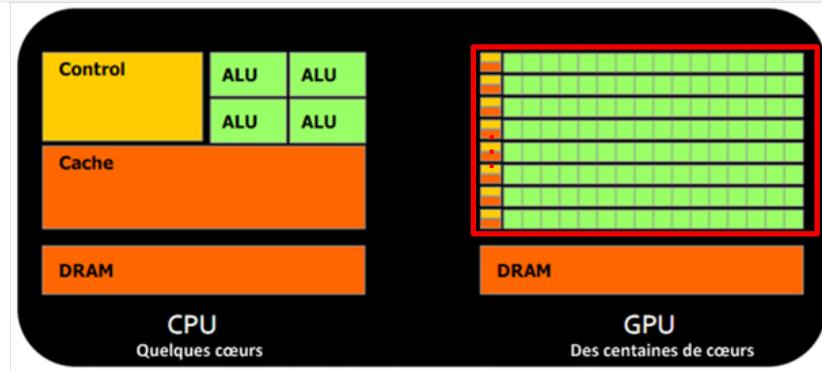
DRAM →

The result of the matrix multiplication is shown as a vector of unknown values. Arrows point from this vector to four separate ALU units, each receiving data from the DRAM box. The ALU units are represented by boxes containing binary strings: 13...2, 21...0, 01...3, and 1-10...1. These strings are color-coded and have some digits highlighted in green, orange, or blue.

**Etape 1** : copies des données dans les différents groupes de travail et allocation mémoire (plus gestion de flux)



# 1) Calcul GPGPU – GPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

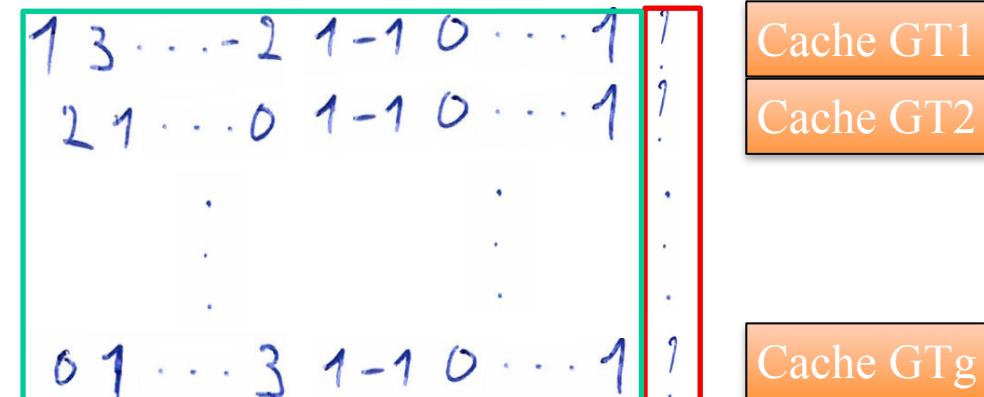
**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

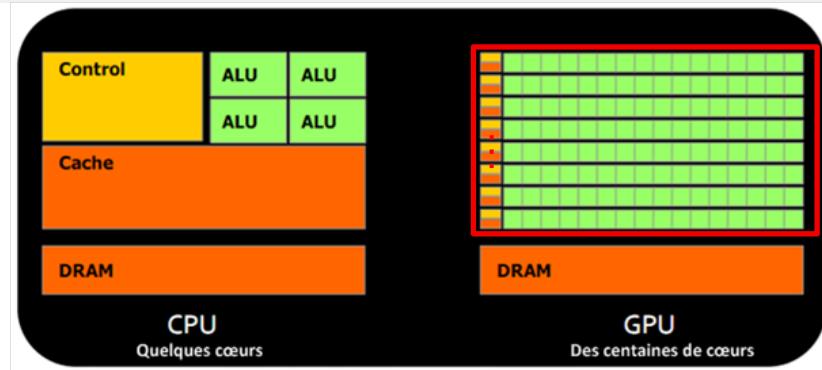
**Control** : Coordonne les ALU et la mémoire

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix} \xrightarrow{\text{DRAM}} 13\dots-221\dots0\dots01\dots31-10\dots1\ 111\dots1$$

**Etape 2** : Toutes les multiplications et additions sont faites en parallèle



# 1) Calcul GPGPU – GPU



**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

$$\begin{pmatrix} 1 & 3 & \dots & -2 \\ 2 & 1 & \dots & 0 \\ -1 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{pmatrix}$$

DRAM

→

1 3 ... -2 2 1 ... 0 ... 0 1 ... 3 1 -1 0 ... 1 ? ? ? ... ?

An orange arrow points from the DRAM box to the result vector, highlighting the final output bits.

**Etape 3** : Résultat copié dans la DRAM

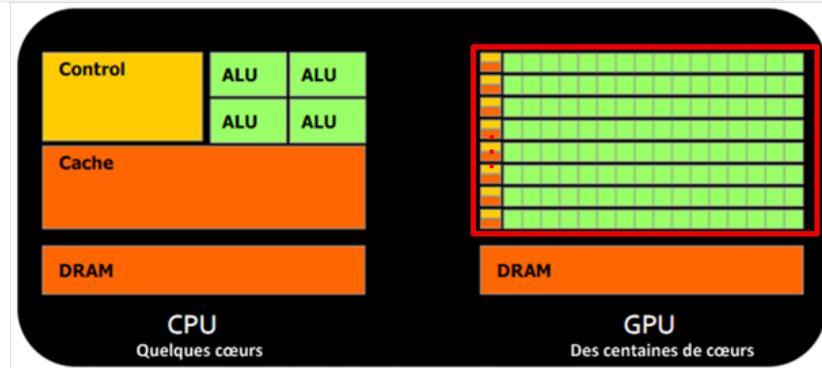
1 3 ... -2 1 -1 0 ... 1  
2 1 ... 0 1 -1 0 ... 1  
.  
. .  
0 1 ... 3 1 -1 0 ... 1

Cache GT1

Cache GT2

Cache GTg

# 1) Calcul GPGPU – GPU



<http://igm.univ-mlv.fr/~dr/XPOSE2013/GPGPU>

**DRAM** : Dynamic Random Access Memory (mémoire classique)

**Cache** : Mémoire (ici) interne au processeur. Rapide mais de taille limité

**ALU** : Arithmetic-Logic Unit. Effectue les calculs.

**Control** : Coordonne les ALU et la mémoire

## Réflexions pour la programmation haut-niveau :

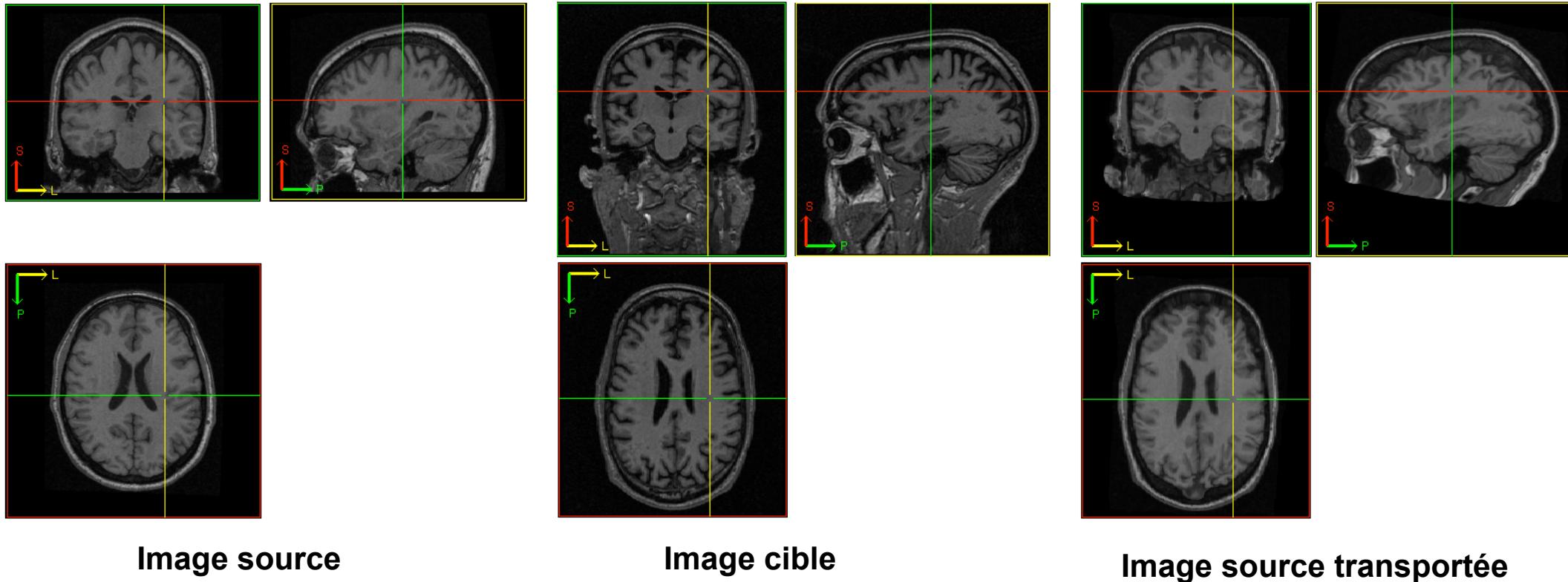
- Est-ce que les mêmes instructions (additions, multiplications, ...) vont s'effectuer sur de gros blocs de données ?
- Est-ce que ces blocs de données sont continus en mémoire ?

## Réflexions pour la programmation bas-niveau :

- [Gain de temps en parallélisant les calculs] - [perte de temps dû aux transferts de données]  $> 0$  ?
- Peut-on rendre négligeable les temps de transferts (latence) ?

# 1) Calcul GPGPU – CPU vs GPU

Recalage d'images médicales 3D avec [Vialard et al., IJCV 2012] :

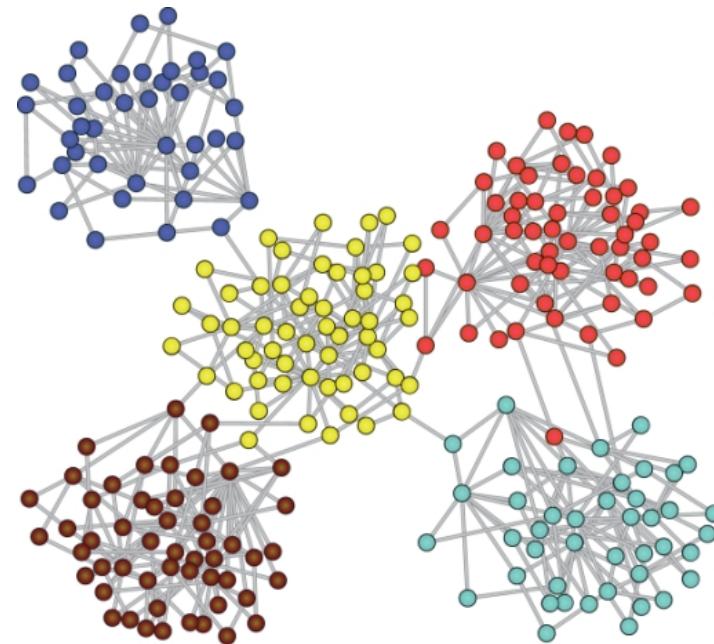


Taille des images	Programme d'origine	Nvidia GTX 780	Intel Iris Graphics 6100
$100 \times 100$	213s	27s	50s
$48 \times 48 \times 48$	58s	1.3s	4.1s
$192 \times 192 \times 192$	1h 51min	2min 8s	17min 38s

# 1) Calcul GPGPU – CPU vs GPU

Partitionnement de graphes principalement issus du 9th DIMACS Challenge ([www.diag.uniroma1.it/challenge9/](http://www.diag.uniroma1.it/challenge9/))

- 1) Graphes sociaux et routiers de grande taille
- 2) Algorithme agglomératif basé sur la modularité [Newman, PNAS 2006]



Exemple de partitionnement de graphe (<https://openi.nlm.nih.gov>)

Computation time (s)	#of clusters	CPU	GPU1	GPU2	GPU3
FACEBOOK - $4 \times 10^3$ nodes	5	0.58	0.081	0.09	0.05
	10	0.12	0.080	0.08	0.06
ZBMATHS - $9 \times 10^4$ nodes	10	0.34	0.27	0.32	0.20
	100	1.50	1.26	1.17	0.88
USA_NY - $3 \times 10^5$ nodes	10	1.71	2.39	1.20	1.55
	100	7.33	8.84	4.43	5.12
	1000	X	X	30.12	32.26
USA_FLA - $10^6$ nodes	10	12.43	10.13	3.73	4.47
	100	70.85	51.67	17.01	23.05
USA_LKS - $3 \times 10^6$ nodes	10	55.84	46.77	10.97	13.53
	50	387.93	175.20	28.38	43.77
	100	X	X	48.24	66.69
USA_USA - $2 \times 10^7$ nodes	2	X	X	185.28	238.42
	10	X	X	213.66	X

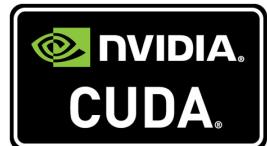
Architectures testées :

- 1) CPU : Intel Core i5-4200h 2.80 GHz
- 2) GPU1 : NVIDIA GeForce 840M
- 3) GPU2 : NVIDIA GTX 780
- 4) GPU3 : NVIDIA Quadro 5000

## 2) En pratique – Installation

### Installation de drivers CUDA :

→ <https://developer.nvidia.com/cuda-downloads>



### Installation de drivers OpenCL :

→ Si le PC a une carte graphique NVIDIA, les drivers OpenCL et CUDA sont là :

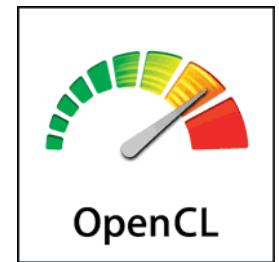
<http://www.nvidia.com/Download/index.aspx?lang=en-us>

→ Si le PC a seulement un processeur Intel, les drivers OpenCL sont là :

<https://software.intel.com/en-us/articles/opencl-drivers>

→ Sur Mac, OpenCL est déjà installé (mais une transition vers Metal est engagée):

<https://developer.apple.com/opencl/>



### Compilateurs possibles :

→ gcc/g++, Clang/xCode, Visual Studio, ...

### Librairies / d'interfaces pour la programmation :

→ Librairie pycuda en Python :

<https://mathematician.de/software/pycuda/>

<https://anaconda.org/lukef/pycuda>

→ Librairie pyopencl en Python :

<https://mathematician.de/software/pyopencl/>

<https://anaconda.org/conda-forge/pyopencl/>

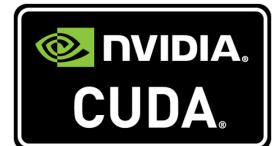
→ Interface Boost en C++ : [http://www.boost.org/doc/libs/1\\_65\\_1/libs/compute](http://www.boost.org/doc/libs/1_65_1/libs/compute)

→ ...

## 2) En pratique – Installation

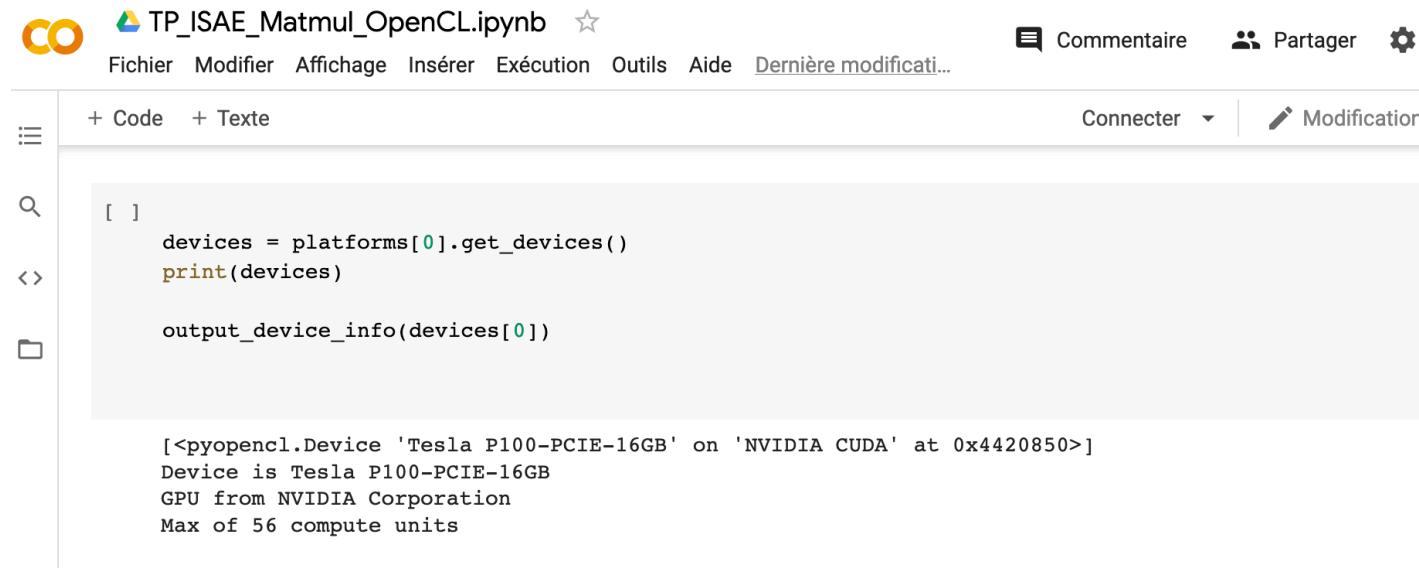
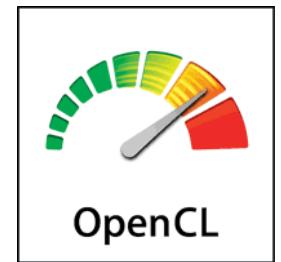
### Remarques (2020) :

- Apple a délaissé *nvidia/cuda*, annonce qu'il ne supportera plus bien longtemps *OpenCL* et pousse sa solution maison *Metal*.
- Tout comme chez *Nvidia* qui supporte *OpenCL*, mais dont les dernières avancées sont sous *CUDA*, *AMD* pousse sa solution maison *ROcm*.



### En vue de faire des TPs :

- Si le PC/Mac a une bonne carte GPU, installer en local le nécessaire.
- Sinon, tout marche très bien sur google colab !



A screenshot of a Google Colab notebook titled "TP\_ISAE\_Matmul\_OpenCL.ipynb". The code cell contains the following Python script:

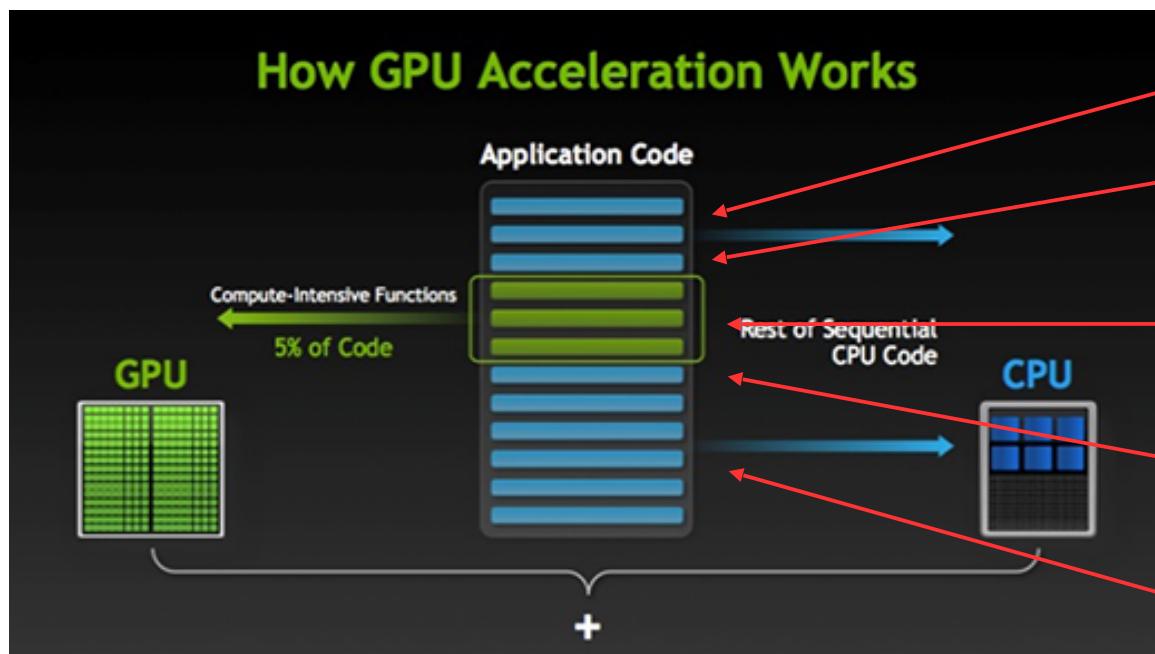
```
[ ]  
devices = platforms[0].get_devices()  
print(devices)  
  
output_device_info(devices[0])
```

The output of the code cell shows the following information:

```
[<pyopencl.Device 'Tesla P100-PCIE-16GB' on 'NVIDIA CUDA' at 0x4420850>]  
Device is Tesla P100-PCIE-16GB  
GPU from NVIDIA Corporation  
Max of 56 compute units
```



## 2) En pratique – Exemple de code opencl appelé par Python



### Dans notre exemple

Code de calcul Python exécuté ;  
Python copie des données dans la mémoire *device* (GPU) ;  
Python appelle une routine OpenCL pour un calcul parallélisé sur les données en mémoire *device* ;  
Python copie les résultats dans la mémoire *host* ;  
Suite de l'exécution du code Python.

## 2) En pratique – Exemple de code opencl appelé par Python

1 : Appel et initialisation de la librairie (API) PyopenCL en début de fichier :

```
import numpy as np
import pyopencl as cl

os.environ["PYOPENCL_CTX"] = '1:0'
os.environ["PYOPENCL_COMPILER_OUTPUT"] = '1'
ctx = cl.create_some_context()

queue = cl.CommandQueue(ctx)
...
...
```

2 : Définition d'une fonction (kernel) qui s'applique en **un point** d'une matrice (array) :

```
prg = cl.Program(ctx, """
__kernel void array2dresampler(
const unsigned int xSize,
const unsigned int ySize,
__global const float *ini_cl,
__global const float *dx_cl,
__global const float *dy_cl,
__global float *rsp_cl)
{
    int rspx = get_global_id(0);
    int rspy = get_global_id(1);
    int inix = rspx+convert_int(dx_cl[rspy+ySize*rspx]+0.5);
    int iniy = rspy+convert_int(dy_cl[rspy+ySize*rspx]+0.5);
    inix=inix*(inix>=0)*(inix<xSize)+(xSize-1)*(inix>=xSize);
    iniy=iniy*(iniy>=0)*(iniy<ySize)+(ySize-1)*(iniy>=ySize);

    rsp_cl[rspy+ySize*rspx] = ini_cl[iniy+ySize*inix];
}
""").build()
```

## 2) En pratique – Exemple de code opencl appelé par Python

### 3 : Transferts DRAM / mémoire GPU et appel de la fonction

```
ini_np=(MovingIm.GetAllData()[:, :]).astype(np.float32)
dfx_np=DispField[:, :, 0].astype(np.float32)
dfy_np=DispField[:, :, 1].astype(np.float32)
mf = cl.mem_flags
ini_cl = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=ini_np)
dfx_cl = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=dfx_np)
dfy_cl = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=dfy_np)
rsp_cl = cl.Buffer(ctx, mf.WRITE_ONLY, ini_np.nbytes)

prg.array2dresampler(queue, ini_np.shape, None, np.int32(ini_np.shape[0]), np.int32(ini_np.shape[1]),ini_cl,dfx_cl,dfy_cl,rsp_cl)

rsp_np = np.empty_like(ini_np)
cl.enqueue_copy(queue, rsp_np, rsp_cl)
```

Définition des arrays dans la mémoire classique (host)

Copie des arrays dans la mémoire GPU (device) et allocation d'une zone mémoire supplémentaire pour le résultat

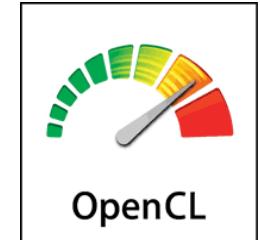
Copie de résultat dans la mémoire classique

Appel de la fonction openCL

## 2) En pratique – Librairies généralistes utilisant du calcul GPGPU

### Algèbre linéaire :

- 1) clBLAS (BLAS niveau 1, 2, 3 – openCL)
- 2) cuBLAS (BLAS niveau 1, 2, 3 – CUDA)
- 3) ViennaCL (openCL)
- 4) VexCL (openCL/CUDA)
- 5) CUSP (CUDA)
- 6) CULA dense et sparse (Lapack et BLAS – CUDA)
- 7) ...



### FFT :

- 1) CUFFT (CUDA)
- 2) clFFT (openCL)
- 3) ...



VexCL



### Génération de nombres aléatoires :

- 1) cuRAND (CUDA)
- 2) clRNG (openCL)

C U S P



clFFT

cuFFT

clRNG



### Numérique généraliste :

- 1) ArrayFire (CUDA et OpenCL)
- 2) ...



## 2) En pratique – Librairies de machine learning utilisant du calcul GPGPU

Deep learning

Caffe



Keras

theano

PYTORCH

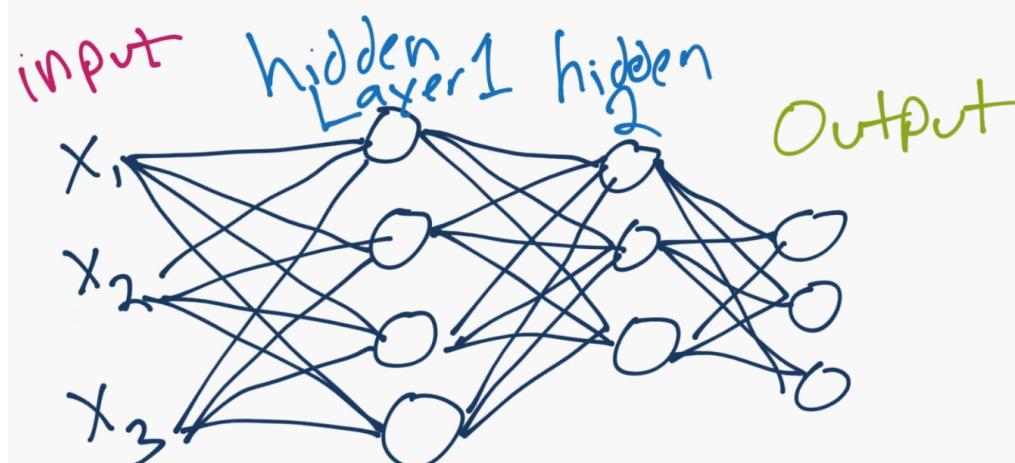


...

Connu

Appris

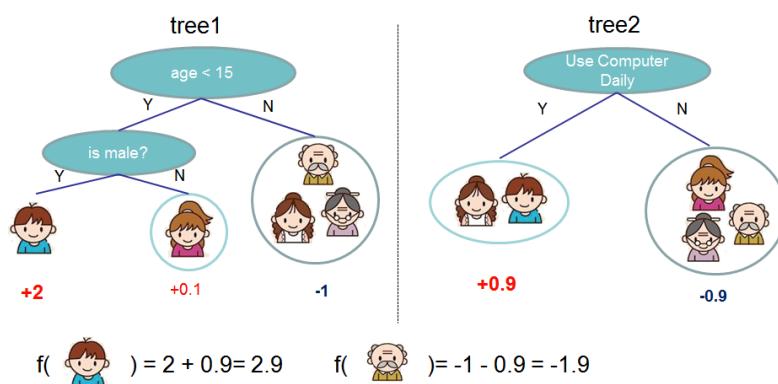
Connu



<https://pythonprogramming.net/neural-networks-machine-learning-tutorial/>

XGBoost

**XGBoost**



- 1) Très efficace sur certains problèmes (signaux, images).
- 2) Apprentissage nécessitant du calcul intensif mais prédiction rapide.
- 3) Dimension du problème d'apprentissage très élevée.
- 4) Nécessité d'avoir énormément de données annotées ou une structure de graphe adaptée aux données.

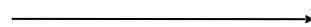
- 1) Méthode basée sur des arbres de décision.
- 2) Très efficace dans le cas général.
- 3) Paramètre à gérer (contrairement à Random Forest) et gros coût calculatoire.

# OpenACC

Directives for Accelerators

```
for (i=0;i<N;i++)
{
    y[i] = 0.0f ;
    x[i] = (float)(i+1) ;
}

for (i=0;i<N;i++)
{
    y[i] = 2.0f * x[i] + y[i];
```



```
#pragma acc data pcreate(x) pcopyout(y)
{
    #pragma acc parallel loop
    for (i=0;i<N;i++)
    {
        y[i] = 0.0f ;
        x[i] = (float)(i+1) ;
    }

    #pragma acc parallel loop
    for (i=0;i<N;i++)
    {
        y[i] = 2.0f * x[i] + y[i];
    }
}
```

- 1) Extrêmement efficace au moins pour l'imagerie et le calcul matriciel.
- 2) Nécessite un investissement de temps non-négligeable pour s'y mettre.
- 3) Efficacité dépendant de l'architecture matérielle mais aussi de la structure des données en mémoire.
- 4) Il existe une multitude de projets liés au calcul GPU.