

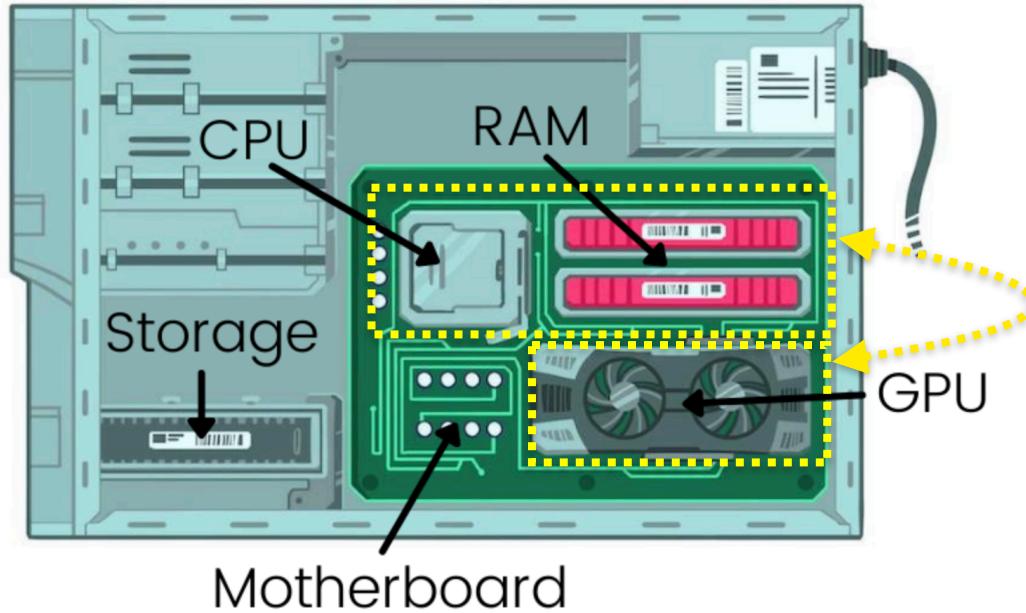
Introduction au calcul GP-GPU

Partie 1 : Mise en contexte

Laurent Risser

Institut de mathématiques de Toulouse
lrisser@math.univ-toulouse.fr

Préambule - Pytorch CPU vs PyTorch GPU



Echanges de données pour une parallélisation massive des traitements intra-réseau de neurones

(Ordre des gains en temps de calculs : facteur 3 à 1000 — typiquement 50)

Mécanisme de transferts similaire à ce qui se fait usuellement avec CUDA :

```
import torch
import torchvision.models as models

import torch.nn as nn

h_resnet18 = models.resnet18()
h_resnet18.fc = nn.Linear(512, 5)
d_resnet18 = h_resnet18.cuda()          ← Copie du réseau de neurones vers le GPU

h_simulated_data = torch.randn([1000,3,64,64])      ← Copie de données vers le GPU
d_mini_batch = h_simulated_data[0:10,:,:,:].cuda()
```

```
d_outputs = d_resnet18(d_mini_batch)

h_outputs = d_outputs.cpu()               ← Copie de la prédition vers la RAM du CPU

print(h_outputs)

tensor([[ 0.2605, -0.4549, -0.5667,  0.8204, -0.4016],
       [ 0.5252, -0.7144, -0.4012,  0.7757, -0.8269],
       [ 0.6739, -0.2131, -1.0644,  1.0779, -0.7419],
       [ 0.7548,  0.0197, -0.2869,  1.0721, -0.6495]],
      grad_fn=<ToCopyBackward0>)
```

Préambule - Pytorch CPU vs PyTorch GPU

Pour les réseaux de neurones :

2 x 36 threads
2 x 2200 euros
en 01/2023

320 tensor cores
7200 euros en
01/2023

Table 2

BERT Inference Performance Comparison between Intel CPU and NVIDIA GPU

	DUAL INTEL XEON GOLD 6240	NVIDIA T4 (TURING)
BERT Inference* Question-Answering (sentences/sec)	2	118
Total Processor TDP	300 W[150x2]	70 W
Energy Efficiency (using TDP)	.007 sentences/sec/W	1.7 sentences/sec/W
GPU Performance Advantage	1.0 (baseline)	59x
GPU Energy-Efficiency Advantage	1.0 (baseline)	240x

(Tests effectués en 2021 —
source : www.curtisswrightds.com)

Apprentissage de réseaux de neurones
RoBERTa sur 400 000 biographies linkedin
pour la prédiction de 28 métiers avec PyTorch.

Serveur CPU IMT : Intel® Xeon® CPU E7-8890 v4 @ 2.20GHz —192 cœurs , 512 Go RAM → **6 heures**

Serveur GPU ANITI : Nvidia Quadro RTX 8000 — 4608 coeurs, 48Go RAM → **1 heure**

Préambule - Pytorch CPU vs PyTorch GPU

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

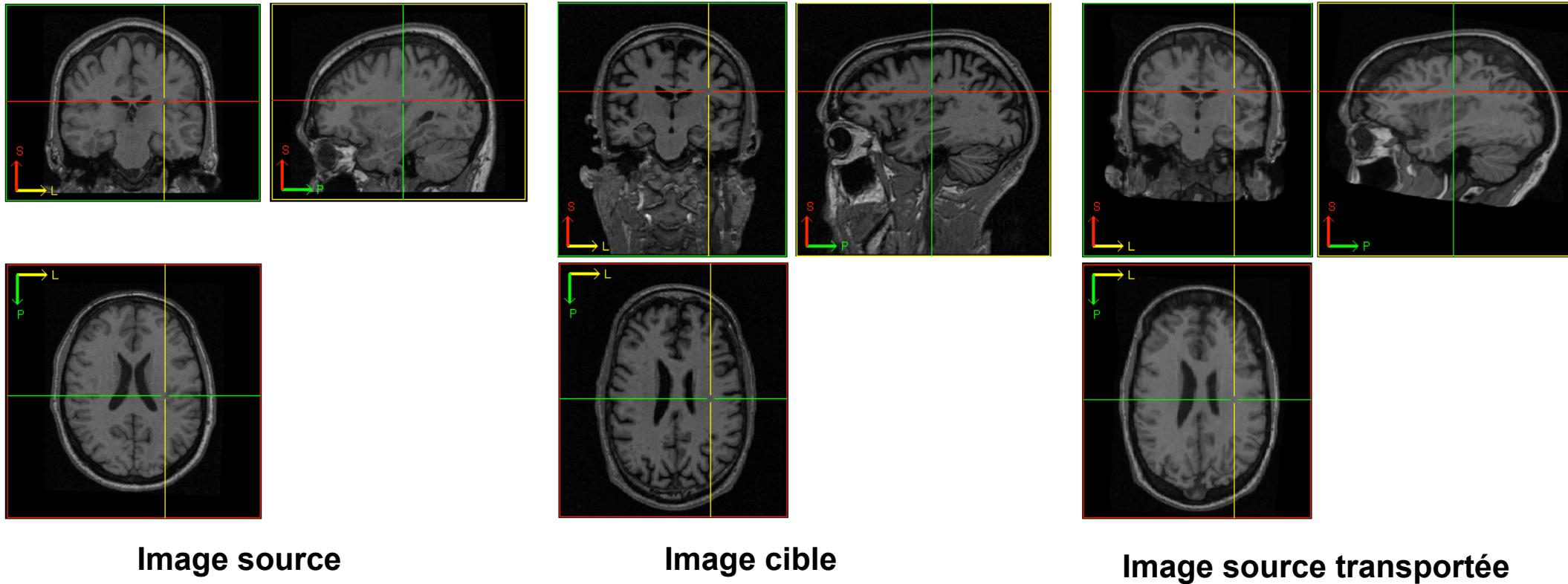


Image source

Image cible

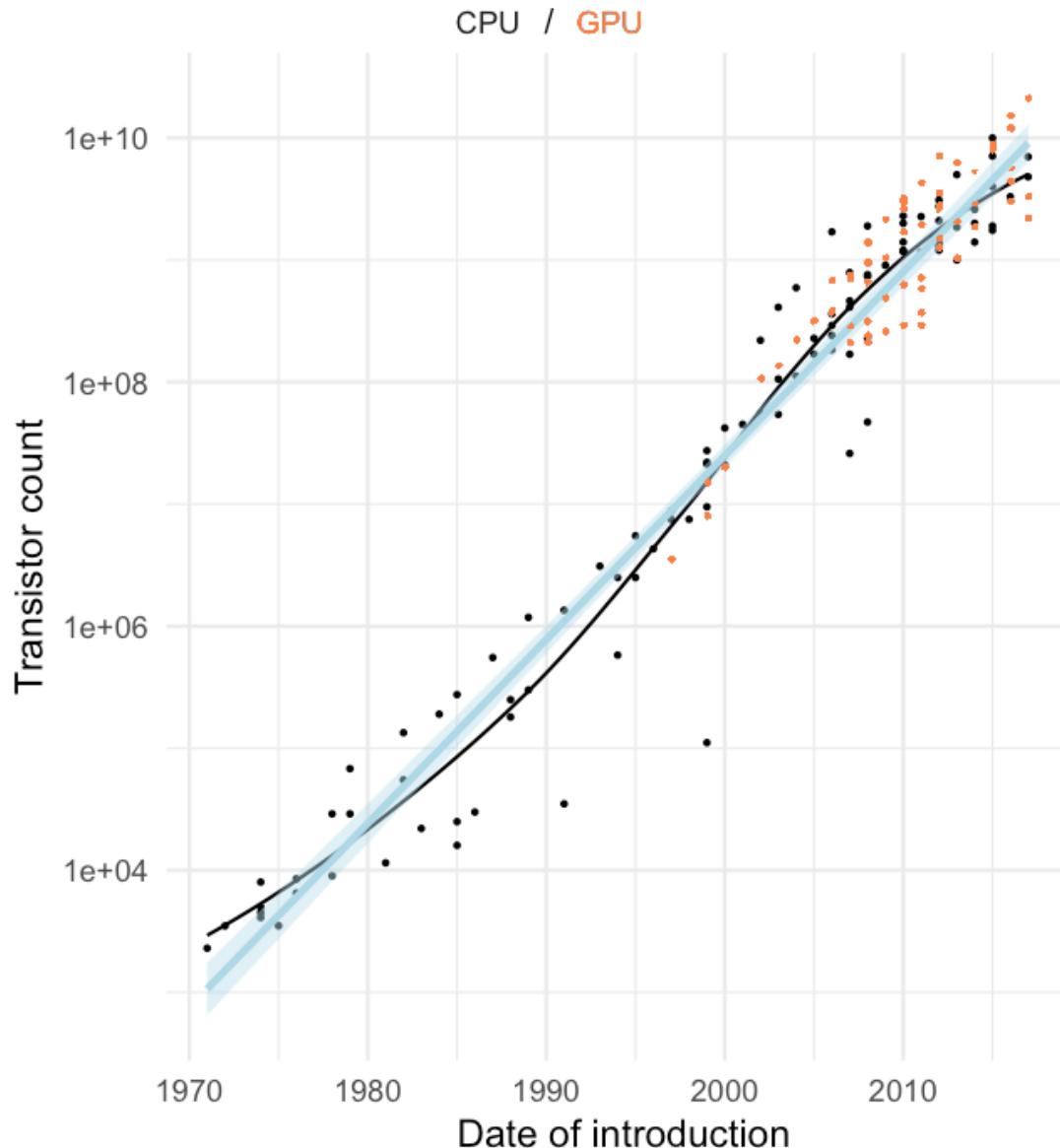
Image source transportée

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

Préambule - loi de Moore

Moore's Law continued

Still linear



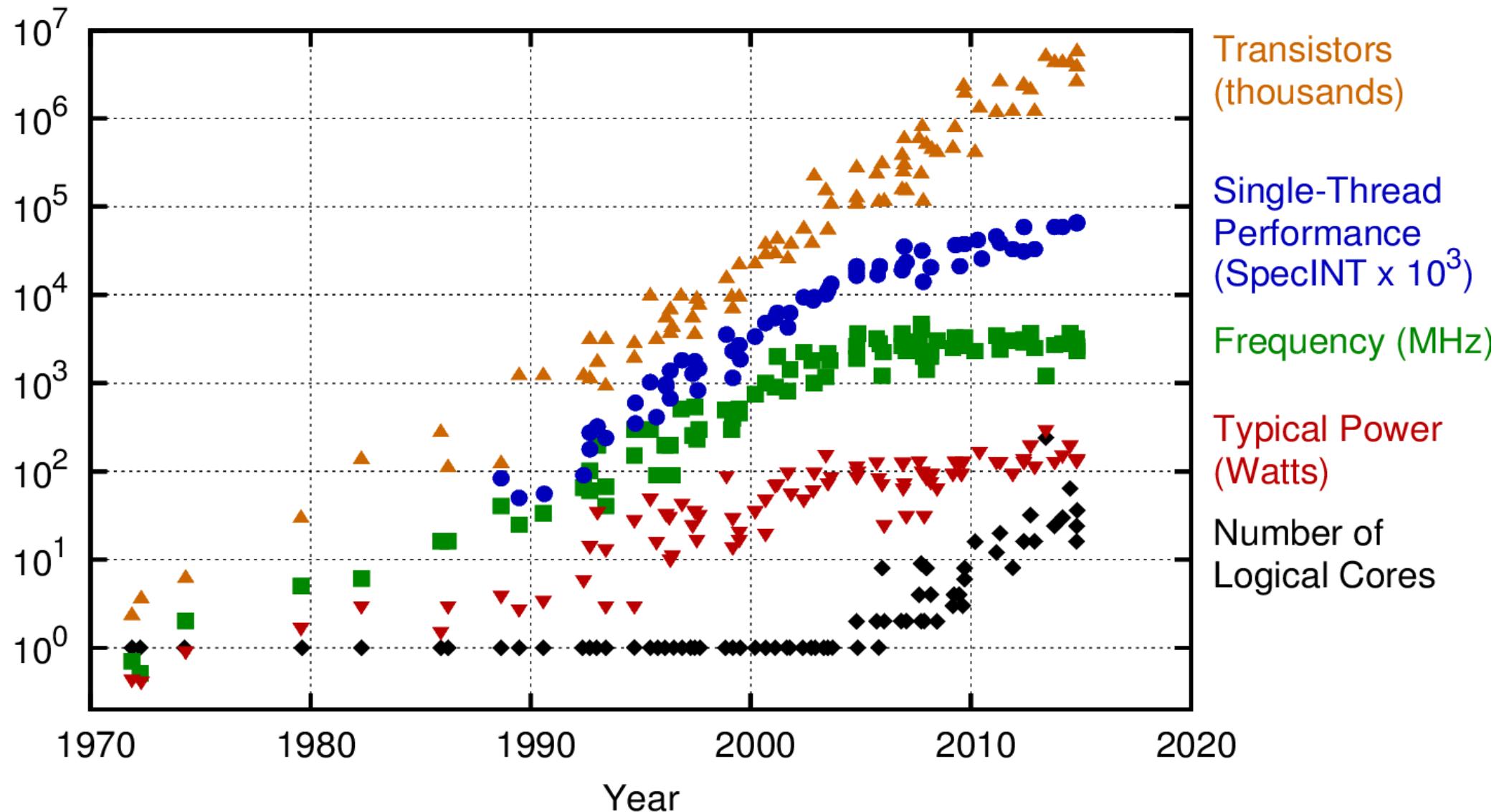
Loi de Moore (fin des années 60)

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

Déclaration de Gordon Moore en 1997

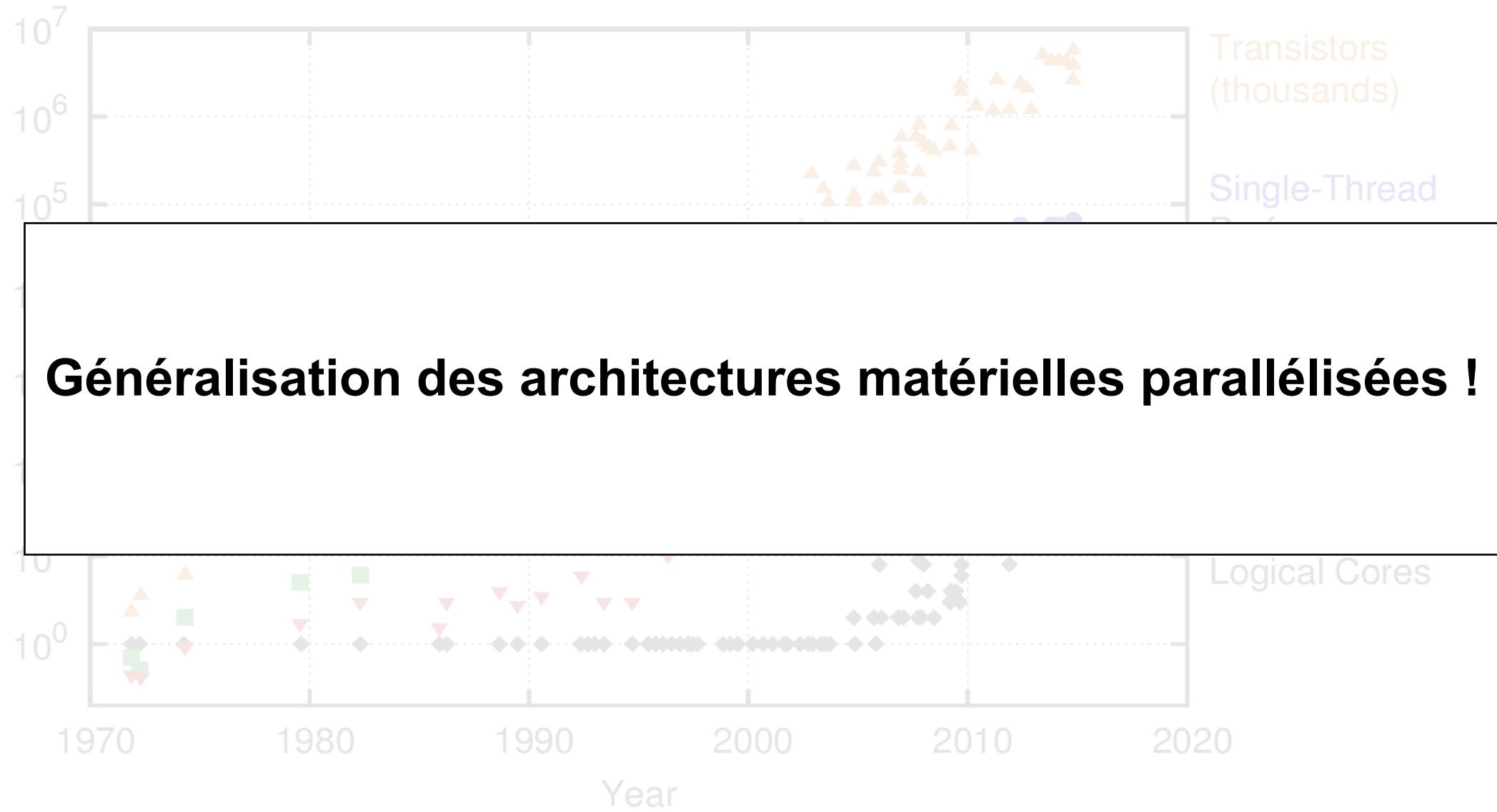
Fin de la loi en 2017 dû à une limite physique.

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

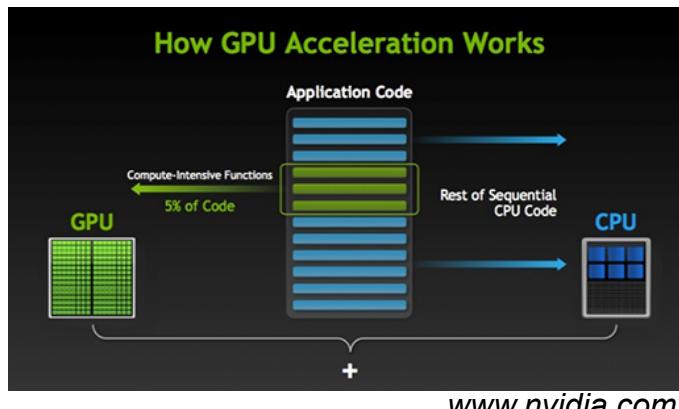
Années 1990 :



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)

2014 - ... : Roc-M (AMD), Metal (Apple), ...

Ces dernières années :

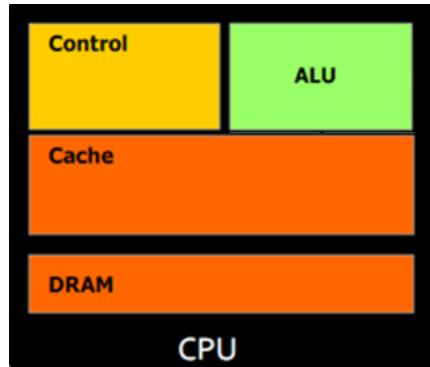
- Utilisation courante du calcul GPU en calcul intensif (HPC)
- Succès du Deep Learning et de XGBoost fortement lié au calcul GPGPU

PREAMBULE

PARTIE 1 : Principes du calcul GPGPU

PARTIE 2 : En pratique

1) Calcul GPGPU – CPU



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

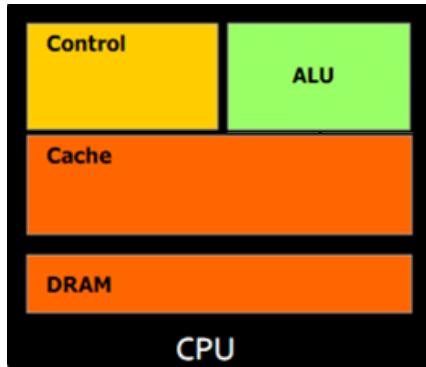
DRAM : Dynamic Random Access Memory (mémoire classique)

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

ALU : Arithmetic-Logic Unit. Unité de 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

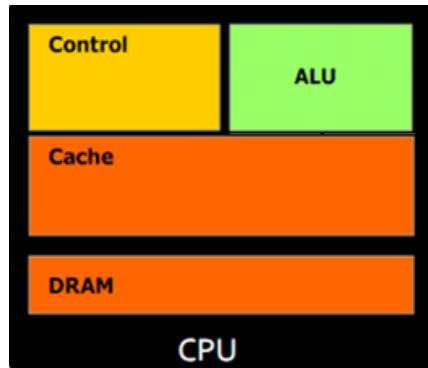
ALU : Arithmetic-Logic Unit. Unité de calculs.

Control : Coordonne les ALU et la mémoire

Illustration : Multiplication matrice / vecteur

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

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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

Illustration : Multiplication matrice / vecteur

The diagram shows a matrix-vector multiplication. On the left, a 4x6 matrix is multiplied by a 6x1 vector. The result is a 4x1 vector. An orange arrow points from the result to a box containing the text: "Structuration possible dans la DRAM ou le Cache".

Matrix (4x6):

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

Vector (6x1):

$$\begin{pmatrix} 1 \\ -1 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{pmatrix}$$

Result (4x1):

$$\begin{pmatrix} ? \\ ? \\ ? \\ ? \end{pmatrix}$$

The diagram shows the components of the multiplication. The matrix is broken down into four 2x3 sub-matrices: a green one (top-left), an orange one (top-right), a light green one (bottom-left), and a blue one (bottom-right). The vector is also shown as four segments: a green one (top), an orange one (middle), a light green one (bottom-left), a blue one (bottom-right), and a red one (far right).

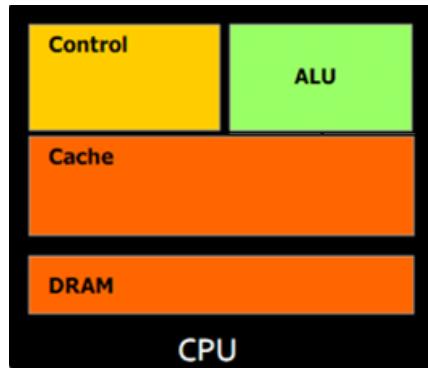
Matrix components:

- Green box: $1 \ 3 \ \dots \ -2$
- Orange box: $2 \ 1 \ \dots \ 0$
- Light green box: $\dots \ \dots \ 0$
- Blue box: $0 \ 1 \ \dots \ 3$

Vector components:

- Green box: $1 \ 3 \ \dots \ -2$
- Orange box: $2 \ 1 \ \dots \ 0$
- Light green box: $\dots \ \dots \ 0$
- Blue box: $1 \ -1 \ 0 \ \dots \ 1$
- Red box: $? \ ? \ ? \ \dots \ ?$

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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

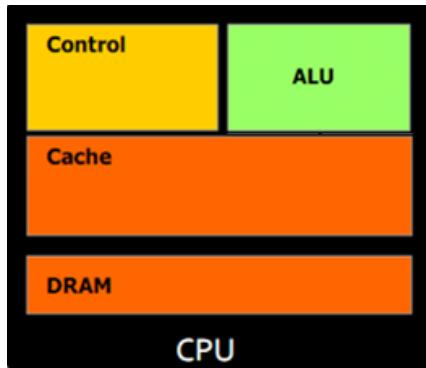
Illustration : Multiplication matrice / vecteur

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

Opérations + et * sur différentes zones mémoires par le CPU

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

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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

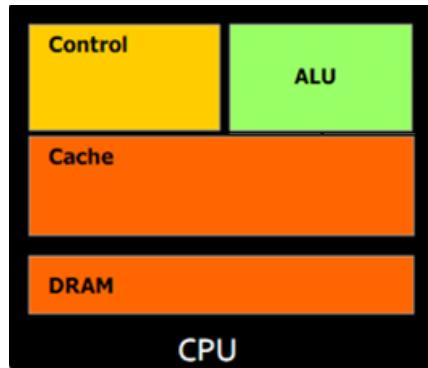
Illustration : Multiplication matrice / vecteur

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

Opérations + et * sur différentes zones mémoires par le CPU

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

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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

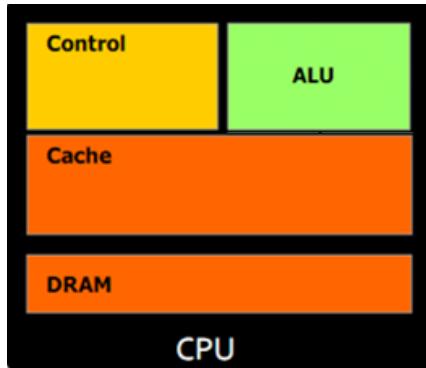
Illustration : Multiplication matrice / vecteur

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

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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

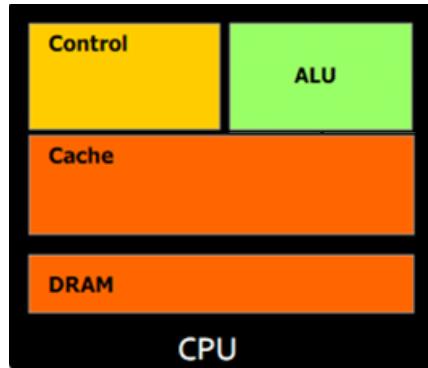
Illustration : Multiplication matrice / vecteur

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

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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

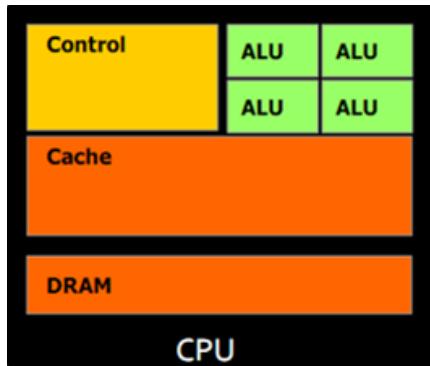
Illustration : Multiplication matrice / vecteur

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

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 – 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. Unité de calculs.

Control : Coordonne les ALU et la mémoire

Illustration : Multiplication matrice / vecteur

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

1er pas vers la parallélisation :
Paradigme *Single Instruction, Multiple Data* (SIMD)

→ L'unité de contrôle demande à plusieurs ALU de faire simultanément le même calcul dans des cases mémoire différentes (notion de WARP)

$$1 \boxed{3 \dots -2} 2 1 \dots 0 \dots 0 1 \dots 3 \boxed{1-1 0 \dots 1} \quad \boxed{\begin{matrix} ? \\ ? \\ ? \\ \vdots \\ ? \end{matrix}}$$

1) Calcul GPGPU – GPU

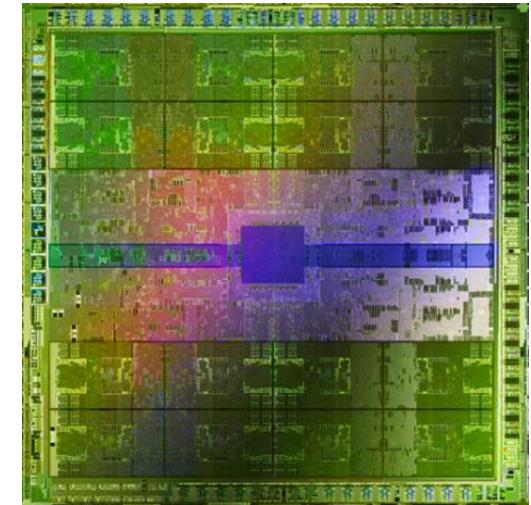
Intuition naïve (et partiellement fausse)

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 :

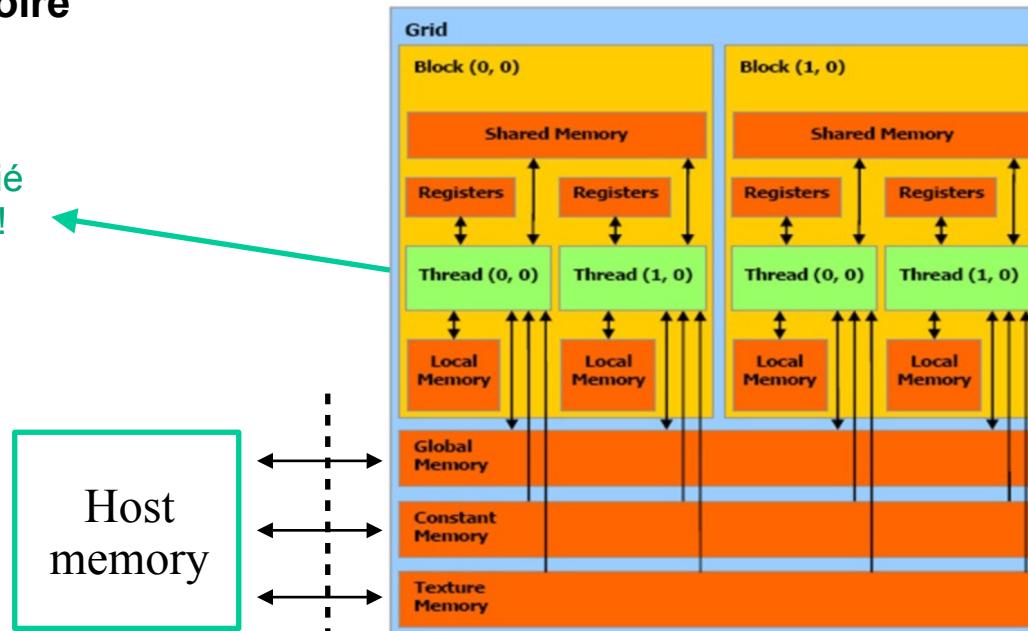
- Contraintes physiques
- Coût des différents niveaux de mémoires (caches L1, L2 et L3)



NVIDIA® Tesla® C2090

Modèle de mémoire

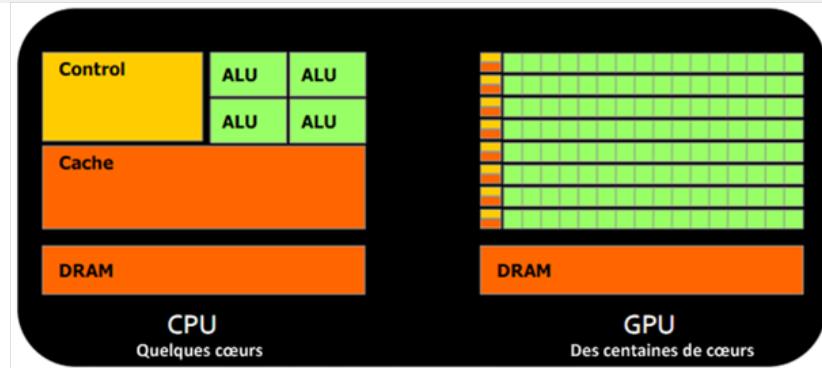
Chaque thread est lié
à une ALU du GPU !



Mémoire
classique
(RAM)

Host
memory

1) Calcul GPGPU – GPU



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

DRAM : Dynamic Random Access Memory (mémoire classique)

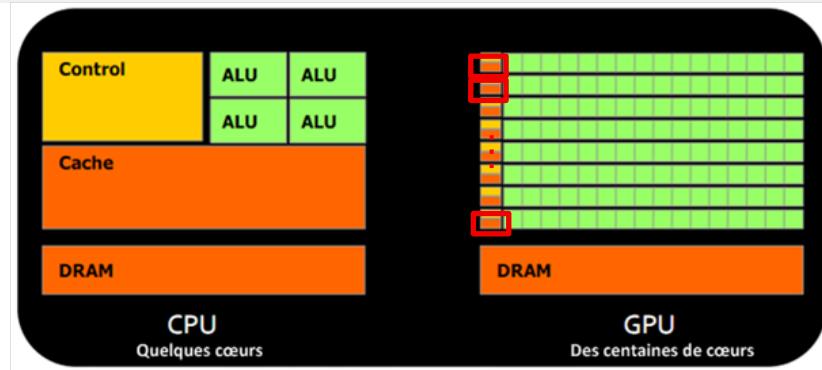
Cache : Mémoire (ici) interne au processeur. Rapide mais de taille limité

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.

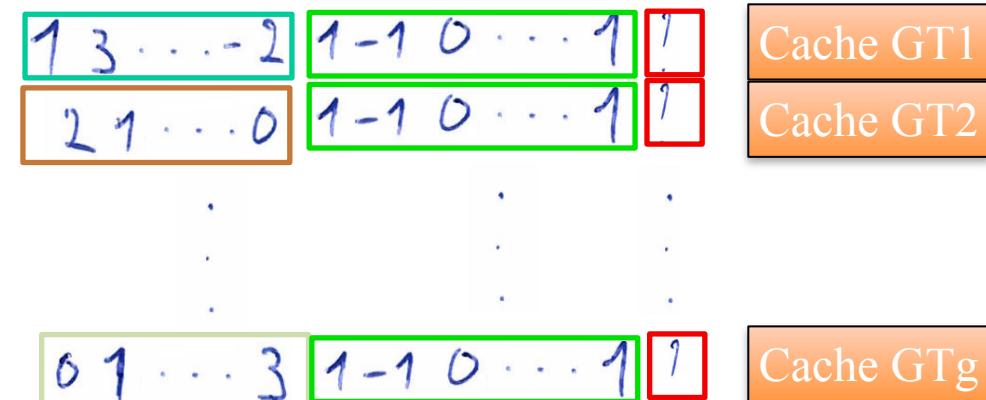
Control : Coordonne les ALU et la mémoire

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

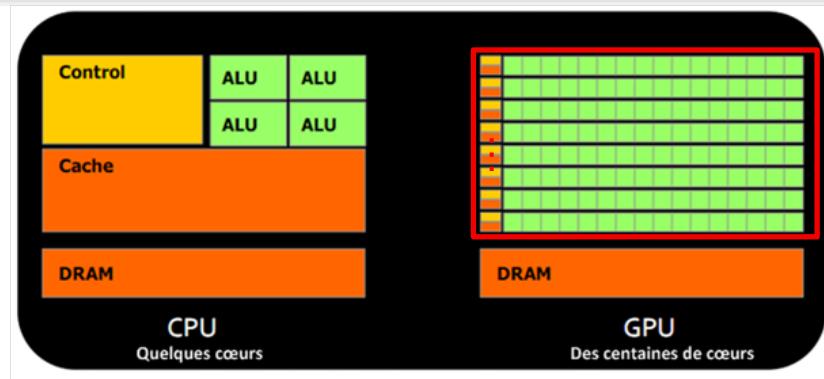
DRAM →

The result of the matrix multiplication is shown as a vector of unknown values. An arrow points from this vector to four separate ALU units. Each ALU receives two inputs: one from the first row of the matrix and one from the vector. The outputs of the ALUs are then combined to produce the final result vector.

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)

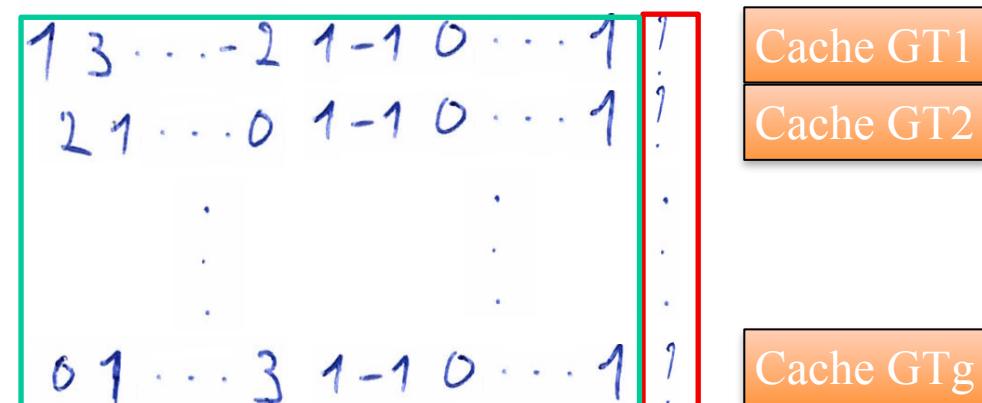
Cache : Mémoire (ici) interne au processeur. Rapide mais de taille limité

ALU : Arithmetic-Logic Unit. Effectue les calculs.

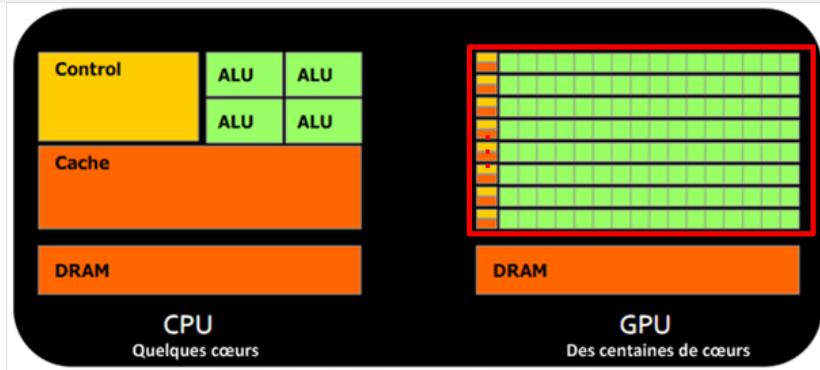
Control : Coordonne les ALU et la mémoire

$$\left(\begin{array}{cccccc} 1 & 3 & \dots & \dots & -2 \\ 2 & 1 & \dots & \dots & 0 \\ -1 & -2 & \dots & \dots & 0 \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ 0 & 1 & \dots & \dots & 3 \end{array} \right) \left(\begin{array}{c} 1 \\ -1 \\ 0 \\ \vdots \\ \vdots \\ 1 \end{array} \right) = \left(\begin{array}{c} ? \\ ? \\ ? \\ \vdots \\ ? \\ ? \end{array} \right) \xrightarrow{\text{DRAM}} 13\dots-221\dots0\dots01\dots31-10\dots1\ 111\dots1$$

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



1) Calcul GPGPU – GPU



<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

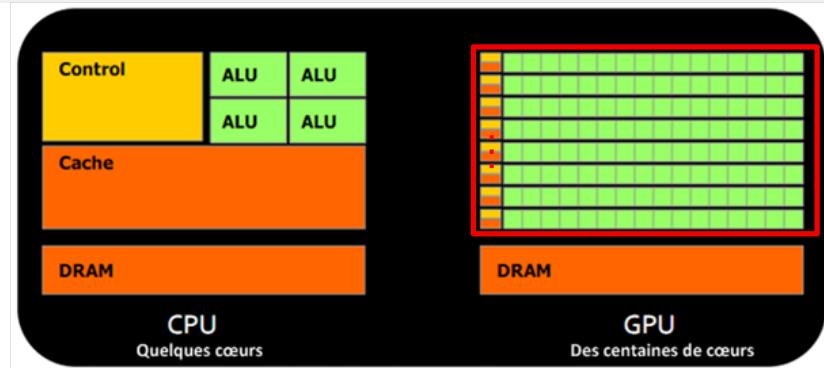
ALU : Arithmetic-Logic Unit. Effectue les calculs.

Control : Coordonne les ALU et la mémoire

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

Etape 3 : Résultat copié dans la DRAM

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

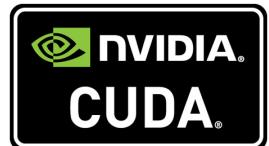
Réflexions pour la programmation haut-niveau :

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

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 ? (→ notion de latence)

2) En pratique – Installation



Installation de drivers CUDA :

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

Compilateurs possibles :

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

Remarques :

- Apple pousse une solution maison *Metal*.
- AMD pousse aussi sa solution maison *ROCM*.
- OpenCL fonctionne normalement sur toutes les cartes mais support limité

2) En pratique – Installation

Librairies d'interfaces PyCuda pour la programmation en Python :

- <https://documentacion.de/pycuda/>
- <https://pypi.org/project/pycuda/>
- <https://anaconda.org/channels/conda-forge/packages/pycuda/overview>
- Installation possible dans des notebooks de Google Colab !



En vue de faire des TPs :

- Si le PC a une carte GPU Nvidia, installer en local le nécessaire.
- Sinon, tout marche très bien sur google colab ou kaggle !

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The top menu bar includes "Fichier", "Modifier", "Affichage", "Insérer", "Exécution", "Outils", and "Aide". The toolbar below has "Commandes", "+ Code", "+ Texte", and "Tout exécuter". The status bar at the bottom shows "Variables", "Terminal", "10:33", and "T4 (Python 3)".

In the main workspace:

- Cell [1]:

```
!pip install pycuda
```

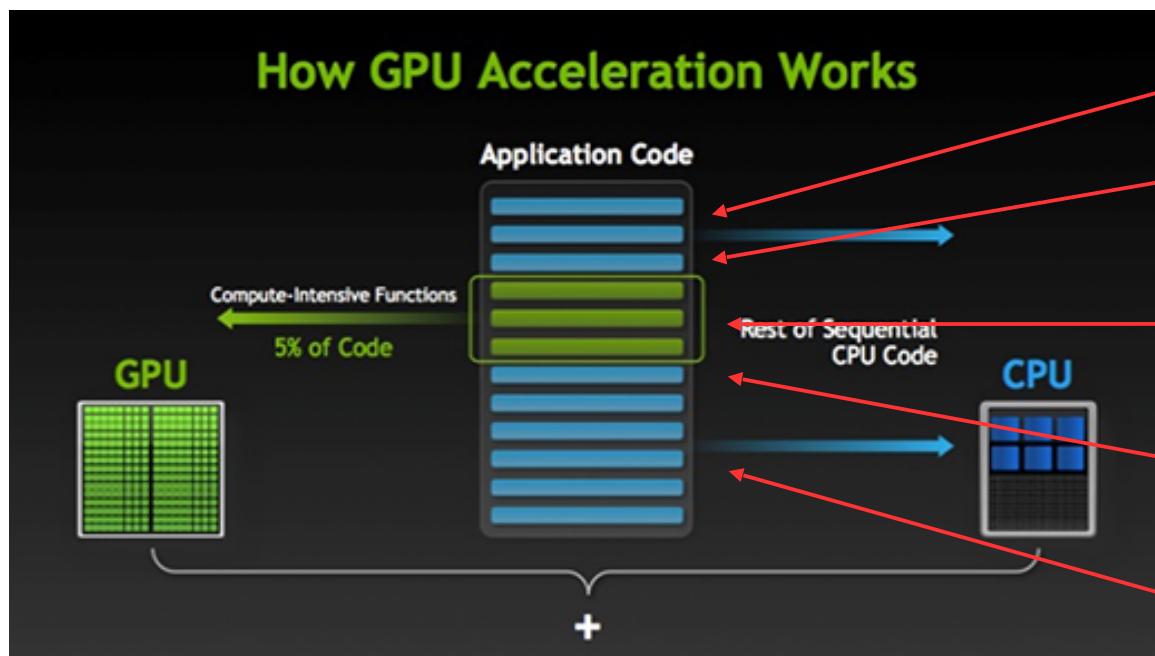
... Collecting pycuda
 Downloading pycuda-2026.1.tar.gz (1.7 MB)
 1.7/1.7 MB 25.8 MB/s eta 0:00:00
Installing build dependencies... done
- Cell [2]:

```
import pycuda
import pycuda.autoinit
```
- Cell [3]:

```
MyDevice=pycuda.driver.Device(0)
print(MyDevice.name())
```

... Tesla T4

2) En pratique – Exemple de code CUDA appelé par Python



Dans notre exemple

Code de calcul Python exécuté ;
Python copie des données dans la mémoire *device* (GPU) ;
Python appelle une routine CUDA 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 CUDA appelé par Python

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

```
import numpy as np
import pycuda.autoinit
from pycuda import driver, gpuarray

from pycuda.compiler import SourceModule
```

2 : Définition et compilation d'une fonction (kernel) qui s'applique en **un point** d'un vecteur :

```
my_kernel="""
__global__ void addition(float *a, float *b, float *result)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    result[index] = a[index] + b[index];
}
"""

mod = SourceModule(my_kernel)

addition = mod.get_function("addition")
```

2) En pratique – Exemple de code CUDA appelé par Python

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

```
a_cpu = np.random.randn(1024).astype(np.float32)
b_cpu = np.random.randn(1024).astype(np.float32)
```

Définition des arrays dans la mémoire classique (host)

```
# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A + B)
c_gpu = gpuarray.empty((1024), np.float32)
```

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

```
addition(a_gpu , b_gpu , c_gpu , block = (1024, 1, 1))
```

Appel de la fonction CUDA

```
c_cpu = gpuarray.GPUArray.get(c_gpu)
```

Copie du résultat dans la mémoire classique

```
print('a = ',a_cpu)
print('b = ',b_cpu)
print('c = ',c_cpu)
```

Vérification du résultat

```
a = [ 1.0868498 -1.161501 -0.20488106 ... -0.65797645 -0.91217256
      1.4827517 ]
b = [ 0.14378278  1.8578753 -0.68487275 ...  2.043415     0.05926542
      -0.82302237]
c = [ 1.2306325   0.6963743 -0.8897538 ...  1.3854387 -0.8529071
      0.65972936]
```

- 1) Le calcul GPU permet de nettement améliorer les temps de calculs lors de l'utilisation de réseaux de neurones par rapport à du calcul CPU standard !
- 2) L'utilisation du calcul GPU peut être relativement simple avec PyTorch ou TensorFlow.
- 3) Afin de bien assimiler ce que peut faire le calcul GPU, saisir ses limites, comprendre ses évolutions, mais aussi savoir comment créer ses propres couches de DNNs, nous allons utiliser CUDA sur des exemples de calculs que l'on retrouve de manière sous-jacente dans les DNNs.