# Introduction aux GPU

David Grimbichler

Université Clermont Auvergne, DSI, Mésocentre

david.grimbichler@uca.fr

# Sommaire

1 Objectifs de ces cours et TP

2 Rappels : architecture des ordinateurs

3 Puissance de calcul

4 Programmation GPU

# Introduction aux GPU

Objectifs :

- Comprendre le fonctionnement d'un GPU moderne
- Identifier les cas où le GPU peut être utilisé
- Connaître les limites de ces calculateurs
- Survoler la programmation haut niveau d'un GPU

# Sommaire

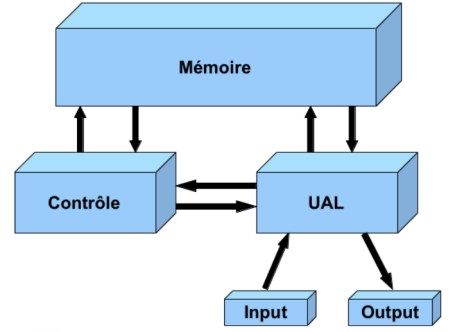
1 Objectifs de ces cours et TP

2 Rappels : architecture des ordinateurs

3 Puissance de calcul

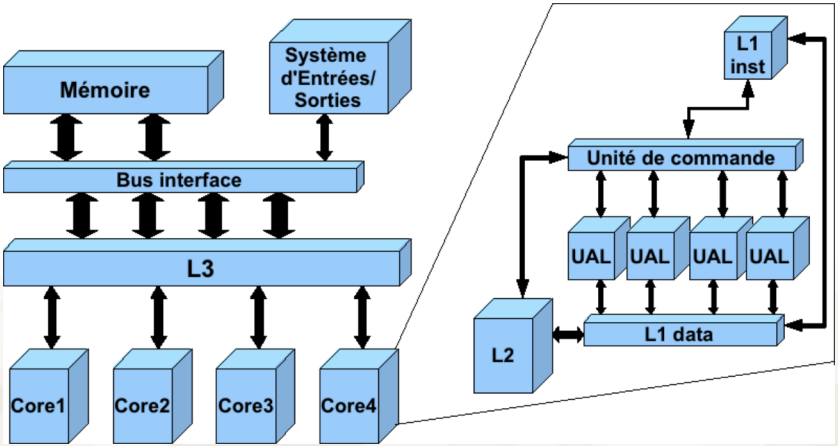
4 Programmation GPU

# Architecture de Von Neumann




- **Mémoire** : contient programme et données
- **Unité arithmétique et logique** : effectue les opérations
- **Unité de contrôle** : séquençage des opérations
- **Unité entrées / sorties**

# Architecture actuelle



# Types et coûts des mémoires

| Type | Taille | Vitesse | Coût/bit |
|---|---|---|---|
| Registre | $< 1$ KB | $< 1$ ns | $$$$ |
| SRAM on-chip | 8 KB - 6 MB | $< 10$ ns | $$$ |
| SRAM off-chip | 1 MB - 16 MB | $< 20$ ns | $$ |
| DRAM | 64 MB - 512 GB | $< 100$ ns | $ |
| Flash | 64 MB - 512 GB | $< 100$ us | c |
| Disk | 64 GB - 1 PB | $< 20$ ms | $\sim 0$ |

Ce que l'on aimerait :

- mémoire la plus grande et la plus rapide possible
- dont la latence est comparable à un temps cycle processeur
- dont la bande passante (nombre d'accès par unité de temps) est très grande

# Pour résumer

Vitesse
Proximité du CPU

Registres

1 ou plusieurs caches

RAM : mémoire primaire

Disques : mémoire secondaire

CD, bandes, ... : mémoire tertiaire

Taille

# Sommaire

1 Objectifs de ces cours et TP

2 Rappels : architecture des ordinateurs

3 Puissance de calcul

4 Programmation GPU

# Calculer plus vite

Besoin de puissance de calcul pour :

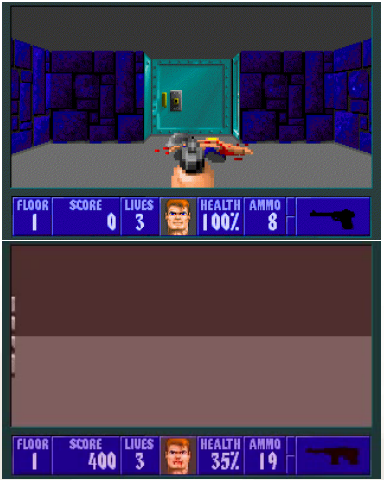
- En 1980 : calculer plus vite pour afficher des choses sur un écran d'ordinateur (générer le signal et faire des calculs en même temps)
- En 1996 : calculer plus vite pour créer de nouveaux jeux vidéos, pour améliorer l'utilisation des logiciels de CAO
- Traitement de problèmes scientifiques constituant de grands challenges, tels que : météo et climat, biologie (génome), géophysique (activité sismique), réactions chimique et réactions nucléaires, ...
- Mais aujourd'hui également pour des applications commerciales : Bases de données parallèles, exploration pétrolière, moteurs de recherche web, réalité virtuelle, suggestions d'achats, ...
- Rendre une meilleure expérience utilisateur, plus fluide, plus rapide (traitement d'images sur les smartphones), plus réactif, ...
- ...

# En 1980

- au début : ordinateurs seulement texte
- puis : mode graphique, tracer des lignes, des cercles, de la couleur
- le CPU (processeur) se chargeait d'effectuer ces opérations (en écrivant dans la mémoire graphique)
- ensuite, arrivée d'accélérateurs 2D : pour tracer une ligne ou un rectangle rempli, on pouvait demander à un autre processeur de le faire (le processeur graphique)

# Mode texte seulement

# En 1992




L'un des premiers jeux 3D (Wolfenstein 3D). Tous les calculs effectués par le CPU :

- son, musique
- gestion des ennemis
- gestion du clavier
- calcul de l'image à afficher (calculs 3D)

Difficile d'effectuer davantage de calculs avec le matériel ⇒ ajout d'un processeur spécifique dédié

# En 1996 : naissance du GPU

Besoins en calculs 3D importants (jeux vidéo, industrie, CAO, ...) :

- développement de cartes spécialisées dans les calculs 3D et l'affichage
- création d'API (bibliothèques) dédiées au pilotage de ces accélérateurs 3D (Glide, OpenGL, Direct3D, Vulkan)
- le CPU est déchargé de ces calculs 3D et peut faire autre chose ⇒ meilleure fluidité

Le GPU :

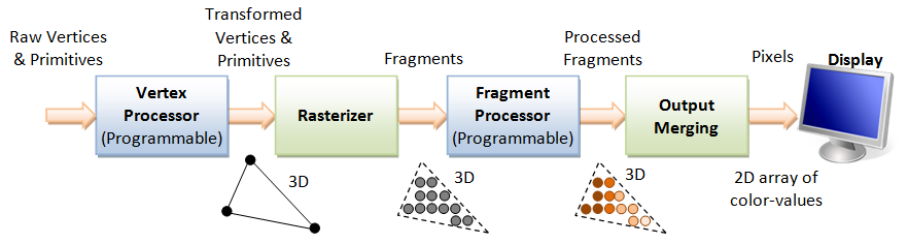
- est une carte qui s'ajoute dans un ordinateur
- branché sur la carte mère
- qui embarque sa propre mémoire (qqs Go)
- génère un signal vidéo
- contient un processeur graphique permettant d'effectuer des calculs 3D

# En 1999

Les GPU font les calculs 3D et l'affichage :

- seules les opérations prévues par le constructeur du GPU sont possibles
- les chercheurs en imagerie aimeraient implémenter d'autres modèles dans les GPU (modèles de lumière, modèles de diffusion, etc.)
- les développeurs de jeux vidéos aimeraient ajouter d'autres effets spéciaux plus jolis et plus complexes à calculer

⇒ évolution des GPU vers davantage de flexibilité en le rendant *programmable*. Le GPU devient donc une carte avec des capacités de *calcul*.

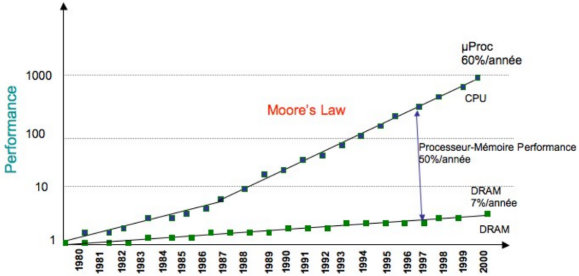


# Comment calculer plus vite ?

Quelques idées pour augmenter la puissance de calcul :

- Améliorer la vitesse du processeur via la fréquence d'horloge (chaleur dégagée)
- Rendre les processeurs plus complexes : exécution simultanée de plusieurs instructions, instructions traitant plusieurs données à la fois
- Évolution des infrastructures vers un haut degré de parallélisme (grand nombre de coeurs)
- Utiliser des processeurs / calculateurs spécialisés, massivement parallèles (les GPU par exemple)
- ...

# Pour calculer plus vite : augmenter la fréquence ?

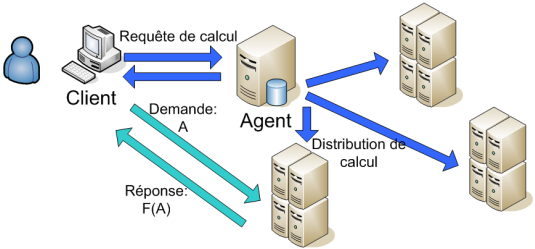



Problématique :

- doubler la fréquence == conso électrique x8
- memory wall : vitesse d'accès à la mémoire augmente beaucoup plus lentement que puissance processeur

⇒ plus de coeurs avec des fréquences réduites (parallélisme massif)

# Pour calculer plus vite : grille de calcul

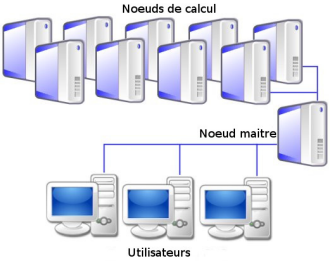



- de nombreux agents indépendants
- hétérogènes (plusieurs OS, plusieurs architectures, ...)
- répartis sur de nombreux sites distants (interconnexion : internet)

Exemples : Seti@Home, Auvergrid, BOINC, EGI, ...

Type de calculs : processus non parallélisés travaillant sur des données découpées (calcul distribué)

# Pour calculer plus vite : cluster de calcul




- noeuds de calcul (puissants) indépendants
- homogènes (OS, applications, voire matériel)
- sur un seul site (interconnexion : 1GbE / 10GbE)

Exemples : Mésocentre Clermont (HPC2), CC-IN2P3, ...

Type de calculs : codes fortement parallélisés (multithread, OpenMP), répartis ou non sur plusieurs noeuds (MPI), ou mixtes.

# Pour calculer plus vite : besoin de parallélisme

Observation :

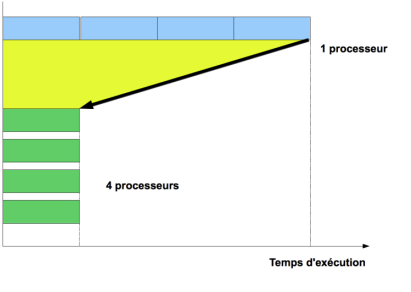
- Traitement de grands volumes de données : découpage des données et calculs sur des lots *en même temps* sur plusieurs ordinateurs ou processeurs
- Simulations avec de grandes dimensions spatiales ou temporelles : découpage de l'espace et/ou du temps et calculs *en même temps* sur plusieurs ordinateurs ou processeurs
- Générer une image constituée 1440x900 pixels (1,2 millions) 30 fois par seconde où chaque pixel résulte d'une équation
- ...

**Paralléliser pour :**

- Traiter des problèmes plus grands et/ou plus complexes.
- Exploiter les architectures actuelles

# Parallélisme

- **Exécutions séquentielles des programmes** : le processeur n'exécute les instructions que l'une après l'autre, en passant éventuellement d'un processus à un autre via des interruptions en simulant le multitâche.
- **Parallélisation** : Ensemble de techniques logicielles et matérielles permettant l'exécution simultanée de séquences d'instructions indépendantes, sur des processeurs différents.



# Séquentiel vs parallèle : calculons... (1)

Tarte aux pommes :

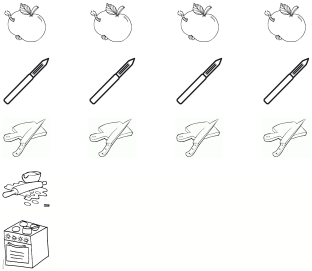
- 4 pommes
- économes pour éplucher (1 minute par pomme)
- couteaux pour découper (1 minute par pomme)
- nécessaire à pâte (5 minutes)
- un moule, un four (30 minutes)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



---

43 minutes à 1 personne

---

Combien à 2 personnes ?

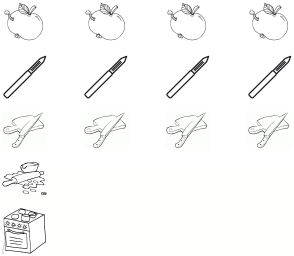
# Séquentiel vs parallèle : calculons... (2)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



37 minutes à 2 personnes

Combien à 4 personnes ?

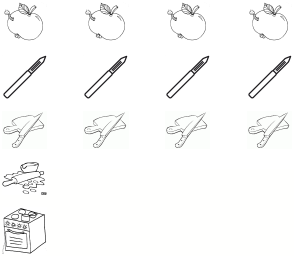
# Séquentiel vs parallèle : calculons... (3)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



35 minutes à 4 personnes

Combien à 3 personnes avec un seul couteau et un seul économe ?

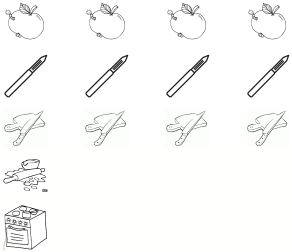
# Séquentiel vs parallèle : calculons... (4)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



---

35 min à 3 + 1 couteau + 1 économe

---

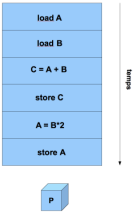
# Classification de Flynn

- Les programmes et les architectures sont classés selon le type d'organisation du flux de données (data) et du flux d'instructions.
- 2 états possibles pour chacune : single ou multiple

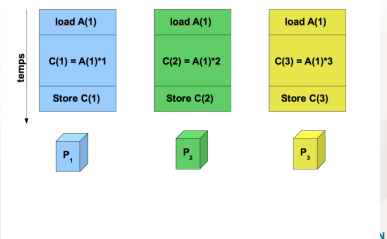
| | |
|---|---|
| **SISD** : Single Instruction, Single Data ⇒ machines séquentielles | **SIMD** : Single Instruction, Multiple Data ⇒ processeurs vectoriels, GPU |
| **MISD** : Multiple Instructions, Single Data ⇒ rare | **MIMD** : Multiple Instructions, Multiple Data ⇒ multiproc, multicores |

# SISD, MISD

**SISD** : architecture séquentielle avec un seul flot d'instructions, un seul flot de données, exécution déterministe.

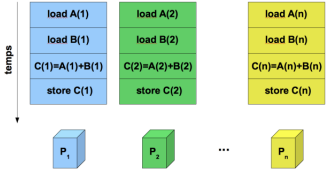



**MISD** un seul flot de données alimente plusieurs unités de traitement, chaque unité de traitement opère indépendamment des autres, sur des flots d'instructions indépendants. Peu implémenté. Ex d'utilisation : exécution en // de plusieurs algos de cryptographie pour le décodage d'1 même message.

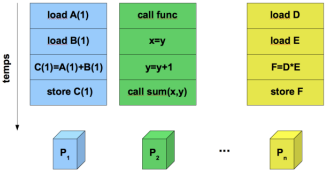


# SIMD, MIMD

**SIMD** architecture // , toutes les unités de traitement exécutent la même instruction à un cycle d'horloge donnée, chaque unité peut opérer sur des données différentes, exécution déterministe.




**MIMD** architecture la plus courante. Chaque unité de traitement peut gérer un flot d'instructions différent. Chaque unité peut opérer sur un flot de données différent. L'exécution peut être synchrone ou asynchrone, déterministe ou non-déterministe.



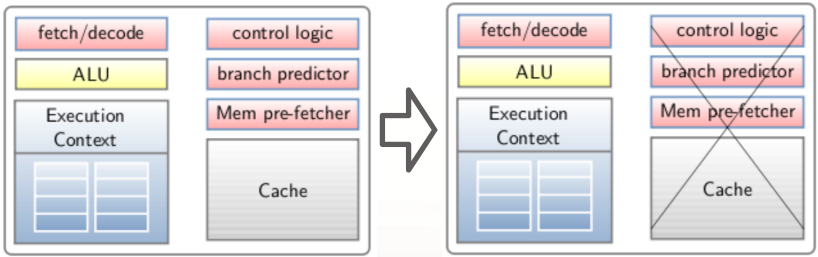
# Et les GPU dans tout ça ?

Comment le constructeur du GPU a répondu à la demande en puissance de calcul programmable ?

- chaque pixel à afficher doit être calculé
- chaque pixel est le résultat d'un ensemble de calculs
- les calculs dépendent de données modélisant ce que l'on veut afficher
- ces données doivent être accessibles par le GPU, dans sa mémoire
- le GPU est branché sur une carte mère avec un processeur principal (le GPU n'est pas autonome)

$\Rightarrow$ le GPU est une machine SIMD.

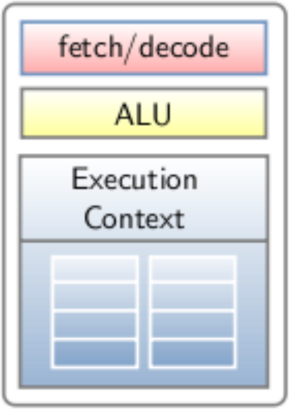
# Comparaison par rapport à un CPU




On enlève les composants qui permettent de gérer le flux d'instructions de façon optimale.

# La base du GPU

On obtient un coeur d'un GPU :

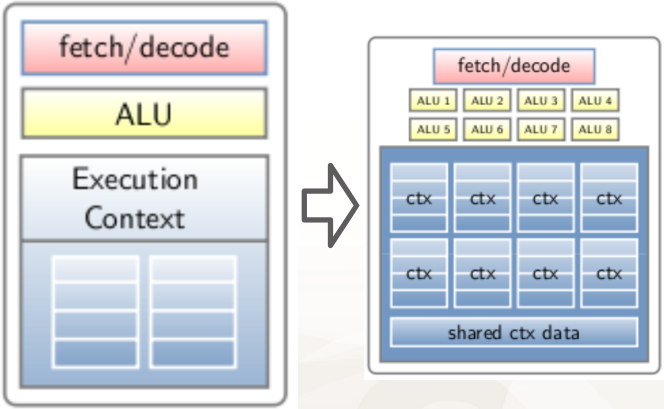


# Base du GPU

On met plein de ces coeurs sur un die (silicone sur lequel sont gravés les transistors) :

# La base du GPU

Partage du flux d'instructions :



# Le GPU

- GPU : ensemble de N petites machines SIMD indépendantes et partageant une mémoire globale (N multiprocesseurs, N=16 ici)
- Un multiprocesseur est 1 petite machine SIMD avec :
  - k ALU synchronisés (ici k=8)
  - 1 décodeur d'instructions
  - 3 mémoires partagées entre toutes les ALU (dont 2 cache)
  - registres distribués entre les ALU
- $16 \times 8 = 128$ coeurs de calcul

# CPU vs GPU

**CPU** : optimisé pour exécuter rapidement une série de tâches de tout type.
**GPU** : optimisé pour exécuter des opérations simples sur de très gros volumes de données.

# Métrique de puissance de calcul

Comment mesurer une puissance de calcul ?

- besoin d'une métrique pour comparer les différentes solutions entre elles
- la métrique doit pouvoir s'appliquer sur tous les matériels visés

$\Rightarrow$ On utilise le FLOP/s ou FLOPS : nombre d'opérations en virgule flottante par seconde (floating-point operations per second) sur une opération de type FMA (fused multiply add : $d = a * b + c$).
Les préfixes Giga- et Téra- sont d'actualité : GFLOPS, TFLOPS.

# Quelques chiffres

Exemple de matériel grand public en 2007 :

| | GeForce 8800 GT | Intel Core 2 Duo |
|---|---|---|
| Nombre de coeurs | 112 | 2 |
| Fréquence | 1,5GHz | 3GHz |
| Performance théorique | 367 GFlops | 32 GFlops |
| Bande passante mémoire | 86,4GB/s | 8,4GB/s |

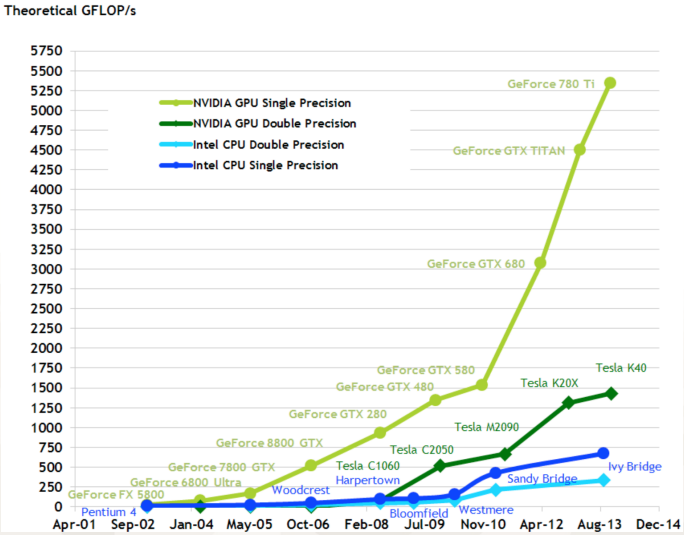
# Quelques chiffres

Exemple de matériel :

| | GeForce GTX 1080 (2007) | Intel Xeon Platinum 8180 (2007) | GeForce RTX 3090 (2020) | AMD Ryzen Threadripper 3990X (2020) |
|---|---|---|---|---|
| Nombre de coeurs | 3584 | 28 | 10496 | 64 |
| Fréquence | 1,5GHz | 3,6GHz | 1,4GHz | 4GHz |
| Performance théorique | 10600 GFlops | 1500 GFlops | 35500 GFlops | 3700 GFlops |
| Bande passante mémoire | 330GT/s | 10,4GT/s | 556GT/s | 24GT/s |
| Prix | 600 euros | 10000 euros | 2500 euros | 4700 euros |

- présent dans beaucoup de PC : marché de masse
- adapté au massivement parallèle (milliers de coeurs)

# CPU vs GPU




À performance égale, les GPU sont :

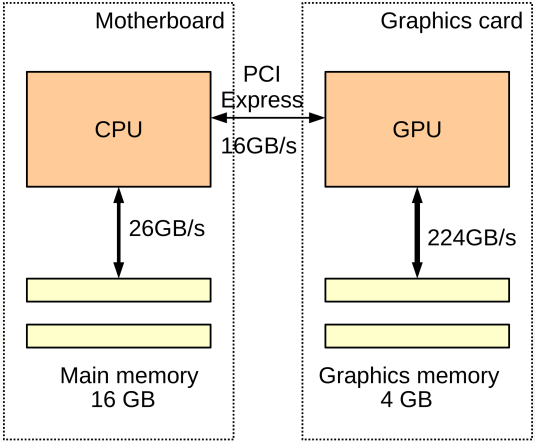
- des solutions plus denses (9x moins de place)
- moins consommateurs d'électricité (7x moins)
- moins chers (6x moins)

# Où trouve-t-on des GPU ?

- Dans les consoles de salon : GPU AMD dans XBox 360/One/Series, Wii, Wii U, PS4, PS5
- Dans les consoles portables : PICA200 dans la 3DS, PowerVR SGX dans la PS Vita, Nvidia GM20B dans la Nintendo Switch, AMD Radeon dans Steam Deck
- Dans les PC et les Mac : AMD Radeon, Nvidia GeForce, Intel HD Graphics
- Dans les smartphones et les tablettes : PowerVR SGX dans les iPhone, Adreno dans les Android à base de Snapdragon, Mali, Vivante, Tegra (Nvidia)
- Dans les SoC (system on chip, type Raspberry Pi) : VideoCore pour les Raspberry Pi, ...
- Dans les processeurs ( !) : les APU AMD, Intel HD Graphics

On les trouve partout $\Rightarrow$ nécessité d'uniformiser leur programmation pour en tirer parti au mieux.

# Transferts




Example configuration:
Intel Core i7 4790, Nvidia GeForce GTX 980

# Conclusion sur les architectures

- Vers une augmentation du **nombre de coeurs** plutôt que de la fréquence d'horloge (problème finesse de gravure, échauffement)
- Programmation de plus en plus complexe
  - nécessité d'une **bonne connaissance des architectures**, notamment au niveau de la mémoire
  - pas de croissance des performances au niveau du coeur $\Rightarrow$ **parallélisation** obligatoire pour gagner des performances
  - Nécessité d'élaborer des **algorithmes** et des programmes capables d'exploiter un grand nombre de processeurs
  - Nécessité d'exploiter le **parallélisme** aux différents niveaux du hardware
  - Programmation sur des architectures hybrides avec différents types de processeurs
- Exploitation des GPU : code extrêmement parallélisé, capable d'utiliser TOUS les threads

# Sommaire

1 Objectifs de ces cours et TP

2 Rappels : architecture des ordinateurs

3 Puissance de calcul

4 Programmation GPU

# Comment qu'on fait alors ?

Ce que l'on sait :

- le GPU a une architecture constituée de multiples coeurs.
- le GPU possède sa propre mémoire .
- le GPU est destiné à des calculs 3D et de l'affichage.
- le GPU est programmable (ie on peut écrire les fonctions qui vont calculer les pixels résultats, appelées *shaders*).
- le GPU reçoit en entrée des données qui seront transformées par les shaders.
- les données reçues par le GPU sont de type coordonnées de points, arêtes, maillage, code des fonctions à calculer, et textures.
- Texture : image envoyée dans la mémoire du GPU qui va être utilisée lors de l'affichage, par exemple en la plaquant sur un quadrilatère. C'est un tableau de nombres.

$\Rightarrow$ si on peut **voir nos données en entrée comme une texture**, alors on peut faire effectuer au GPU un calcul arbitraire sur ces données via les shaders.

Exemple : en entrée 2 matrices 256x256 vues comme des textures float + le code du produit matriciel écrit dans un shader.

# Comment le parallélisme fonctionne sur le GPU ?

Dans un langage haut niveau orienté calculs 3D comme GLSL, le GPU :

- reçoit en entrée les sommets (les *vertex*) et les arêtes (les primitives : triangles, rectangles, lignes, ...)
- les traite en parallèle dans un premier programme (le *vertex shader*)
- puis chaque pixel couvert par les primitives est traité en parallèle dans un second programme (le *fragment shader*)

⇒ C'est ce *fragment shader* qui nous intéresse pour les calculs : pour chaque pixel à calculer dans l'image résultat, ce *shader* est exécuté par le GPU dans un thread (dans un coeur), en parallèle selon les ressources propres du GPU.
Par exemple : avec un GPU disposant de 256 coeurs et une image 256x300 pixels à remplir, 300 lots de 256 threads en parallèle vont exécuter le shader, un lot pour chaque ligne de l'image (contenant 256 pixels).

# TP1 : Comparaison calcul CPU vs GPU

Objectifs du TP :

- Mesurer les temps d'exécution du même code simple sur CPU et GPU
- Voir un exemple de programmation GPU haut niveau en Javascript
- Travailler sur un exemple de programmation GPU haut niveau en GLSL pour générer des images et comprendre le parallélisme GPU
- Constater les limites et les comprendre

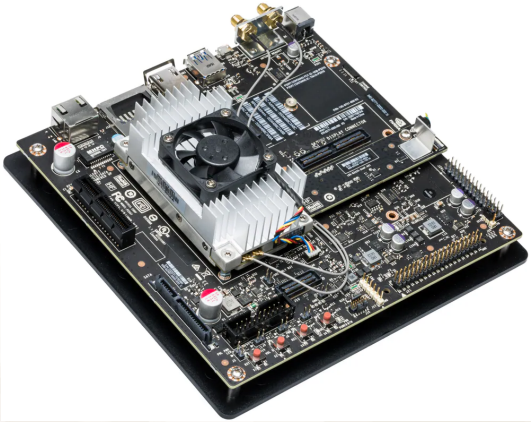
Pour cette première série de TP, possibilité d'utiliser votre ordinateur portable personnel ou une carte Nvidia Jetson TX2.

Fichiers à récupérer pour le TP : `https://drive.uca.fr/f/94d516be04744da589f4/`

# TP1 : Mise en place des cartes Jetson TX2

Branchements :

- 2 antennes WiFi
- 1 câble ethernet
- 1 adaptateur USB, clavier, souris
- câble HDMI, écran
- cordon secteur, alimentation
- Bouton démarrage



# TP1 : Exercice 1

Exercice 1 : comparaison simple CPU vs GPU

Côté CPU :

- Parmi les fichiers du TP, rendez-vous dans `TP1/01-benchmark/python/` .
- Ouvrez le fichier `matmul01.py` avec votre éditeur favori.
- Lisez et comprenez de quoi il est question dans ce code.
- Observez la mesure de temps vers la fin du fichier et comprenez son intérêt. Pourquoi ne pas utiliser la commande linux `time` qui permet de compter le temps écoulé d'un programme ?
- Complétez la fonction `matrix_multiply` vers la ligne 55
- Vérifiez son bon fonctionnement.
- Prenez quelques mesures du temps d'exécution pour des tailles de matrice croissantes ; par exemple pour des matrices 8x8, 32x32, 128x128, 256x256, 512x512, 1024x1024, 2048x2048, 4096x4096, 8192x8192, . . .

# TP1 : Exercice 1

Exercice 1 : comparaison simple CPU vs GPU

Côté GPU :

- Parmi les fichiers du TP, rendez-vous dans `TP1/01-benchmark/gpu/` .
- Ouvrez le fichier `matmulgpu01.html` avec votre éditeur favori.
- Lisez et comprenez de quoi il est question dans ce code.
- Observez la mesure de temps vers la ligne 40, et la multiplication matricielle vers la ligne 28.
- Exécutez ce code dans votre navigateur, et soyez attentifs aux messages de la console de développement (F12).
- Prenez quelques mesures du temps d'exécution pour des tailles de matrice croissantes ; par exemple pour des matrices 8x8, 32x32, 128x128, 256x256, 512x512, 1024x1024, 2048x2048, 4096x4096, 8192x8192, . . .

Quelle est votre conclusion ?

# TP1 : Exercice 1

Exercice 1 : comparaison avec d'autres bibliothèques

Récupérez : `https://drive.uca.fr/f/2967020172df4ecabc73/`

- Ouvrez les fichiers python et regardez de quoi il s'agit.
- Lancer en root (sudo) les commandes `tegrastats` et `jtop` .
- Pour chaque code python de cette archive, vérifiez si le calcul utilise le GPU ou le CPU.
- Comment les performances se comparent entre elles ? Et par rapport à votre implémentation à la main précédente ?

Quelle est votre conclusion ?

# TP1 : Cadre

Utilisation du site web `https://www.shadertoy.com/new` :

- Se focaliser sur le code qui va tourner sur le GPU et masquer les autres difficultés de mise en place
- utilisation du langage GLSL (OpenGL Shading Language), langage des shaders en OpenGL
- Le code de votre shader va tourner en multithread sur le GPU, **un thread pour chaque pixel de sortie**
- Le shader a donc une seule donnée produite : la couleur du pixel sur lequel le thread actuel travaille.
- Le shader peut avoir des entrées : prédéfinies (temps, taille de l'image, clic souris, ...), variables (image webcam, image vidéo, ...)

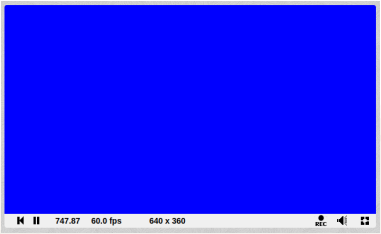
# TP1 : Langage GLSL

- Types de données :
  - float, vec2, vec3, vec4
  - int, ivec2, ivec3, ivec4
  - bool, bvec2, bvec3, bvec4
  - mat2, mat3, mat4
  - void, sampler2D
- Paramètres des fonctions : in, out, inout, const
- Composantes des vecteurs : {x,y,z,w} ou {r,g,b,a}
- Fonctions intégrées : trigo (sin, cos, ...), exponentielles (pow, exp, log, sqrt, ...), logiques (abs, min, max, ceil, floor, clamp, fract, mix, mod, sign, step, smoothstep, ...)
- etc.
- Voir par exemple : GLSL Cheatsheet `http://mew.cx/glsl_quickref.pdf`

# TP1 : Exercice 2

## Exercice 2 : une image couleur unie

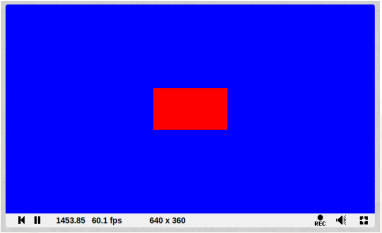
```
void mainImage( out vec4 fragColor,
                in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Blue pixel color
    vec3 col = vec3(0.0, 0.0, 1.0);

    // Output to screen
    fragColor = vec4(col,1.0);
}
```



# TP1 : Exercice 3

## Exercice 3 : un rectangle au centre de l'image

```
void mainImage( out vec4 fragColor,
                in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    vec3 col = vec3(0.0, 0.0, 1.0); // Blue
    if( CURRENT PIXEL IS IN RECTANGLE )
        col = vec3(1.0, 0.0, 0.0); // Red

    // Output to screen
    fragColor = vec4(col,1.0);
}
```



# TP1 : Exercice 4

Exercice 4 : d'autres formes

1. Modifiez le code pour afficher un cercle, une ellipse
2. Même chose avec une image bicolore gauche/droite
3. Même chose avec une ligne verticale noire sur la séparation
4. Même chose avec une séparation se déplaçant dans le temps
5. Même chose avec une séparation verticale à l'endroit du clic souris

# TP1 : Exercice 5

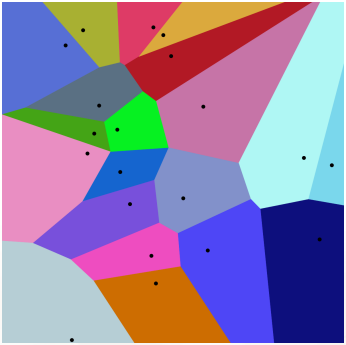
Exercice 5 : Image de Voronoï
Diagramme de Voronoï : découpage du plan en cellules à partir d'un ensemble discret de points appelés germes. Chaque cellule enferme un seul germe, et forme l'ensemble des points du plan plus proches de ce germe que de tous les autres.

Dans le plan affine $\mathbb{R}^2$ : soit $S$ un ensemble fini de $n$ points du plan. Les éléments de $S$ sont appelés centres ou germes. On appelle cellule de Voronoï associée à un élément $p$ de $S$ l'ensemble des points qui sont plus proches de $p$ que de tout autre point de $S$ :

$$Vor_S(p) = \left\{ x \in \mathbb{R}^2 / \forall q \in S, \|x - p\| \leq \|x - q\| \right\}$$

# TP1 : Exercice 5

Exercice 5 : Diagramme de Voronoï, sur GPU

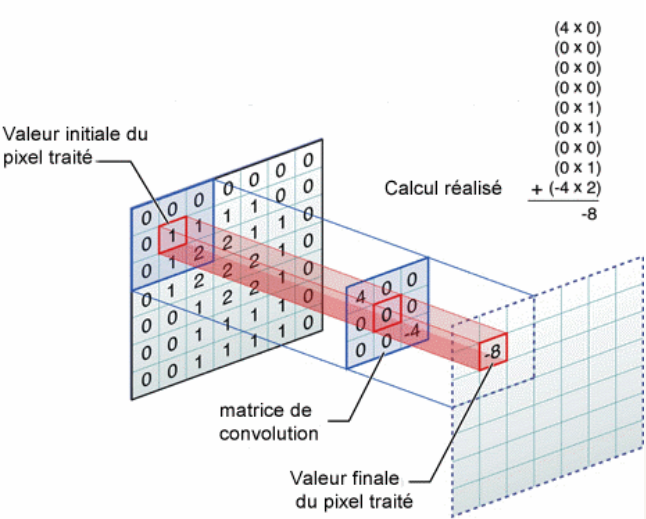


Exercice : implémenter sur GPU en GLSL la génération de l'image de Voronoï des points présents dans le fichier `TP1/02-glsl/voronoi01.txt` avec les couleurs des cellules associées.

# TP2 : Convolution avec OpenCV

```python
import cv2
import numpy as np
import time

image = cv2.imread('test.jpg')
if image is None:
  print('Could not read image')

kernel1 = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])

cv2.imshow('Original', image)
prev_frame_time = 0
new_frame_time = 0
while true:
  filtered = cv2.filter2D(src=image, ddepth=-1, kernel=kernel1)
  new_frame_time = time.time()
  fps = 1/(new_frame_time-prev_frame_time)
  prev_frame_time = new_frame_time
  cv2.imshow('Filtered (%.2f fps)' % fps, filtered)
  if cv2.waitKey(1) & 0xFF == ord('q'):
    break

cv2.destroyAllWindows()
```

# TP2 : Filtres de convolution



# TP2 : Convolution sur GPU

Exercice 2 : implémenter, tester, comparer les filtres suivants :

$$F_1 = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

$$F_2 = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$F_3 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$F_4 = \begin{pmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

$$F_5 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

$$F_6 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

$$F_7 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$