Techniques de calcul haute performance sur GPU

Marc Morère – marc.morere@ipsa.fr

Calcul Parallèle

- Pourquoi et quand paralléliser : performance, énergie, latence vs débit
- Architectures : CPU multicœurs, GPU, SIMD, clusters
- Choisir un modèle de programmation :
  - partagée vs distribuée,
  - data vs task
- Identifier les goulets d'étranglement (Amdahl, déséquilibre, latences mémoire)
- Ecrire/évaluer de petits programmes (OpenMP, MPI, CUDA, Vulkan, Python, C++)

Calcul Parallèle

- Fin de la course au GHz, murs consommation/thermique
- Parallélisme partout : smartphones, navigateurs, IA, serveurs
- Définitions
  - Latence vs débit
  - Scalabilité forte vs faible

Architectures

- CPU
  - Multi-cœurs, SMT (Simultaneous Multi-Threading) / Hyper-Threading
  - SIMD :
    - SSE (X86) : Streaming SIMD Extensions (ajout de nouveaux registres 128 bits)
    - AVX (X86) : Advanced Vector eXtensions, AVX2, AVX-512
    - NEON (ARM) : 32 registres 64 bits ou 16 de 128 bits
- GPU
  - SIMT (Single Instruction Multi Thtreads) : milliers de threads légers
  - Warps / Wavefronts (groupes de threads)
- Clusters
  - Interconnexion réseau
  - Stockage distribué

Hiérarchie mémoire & performance

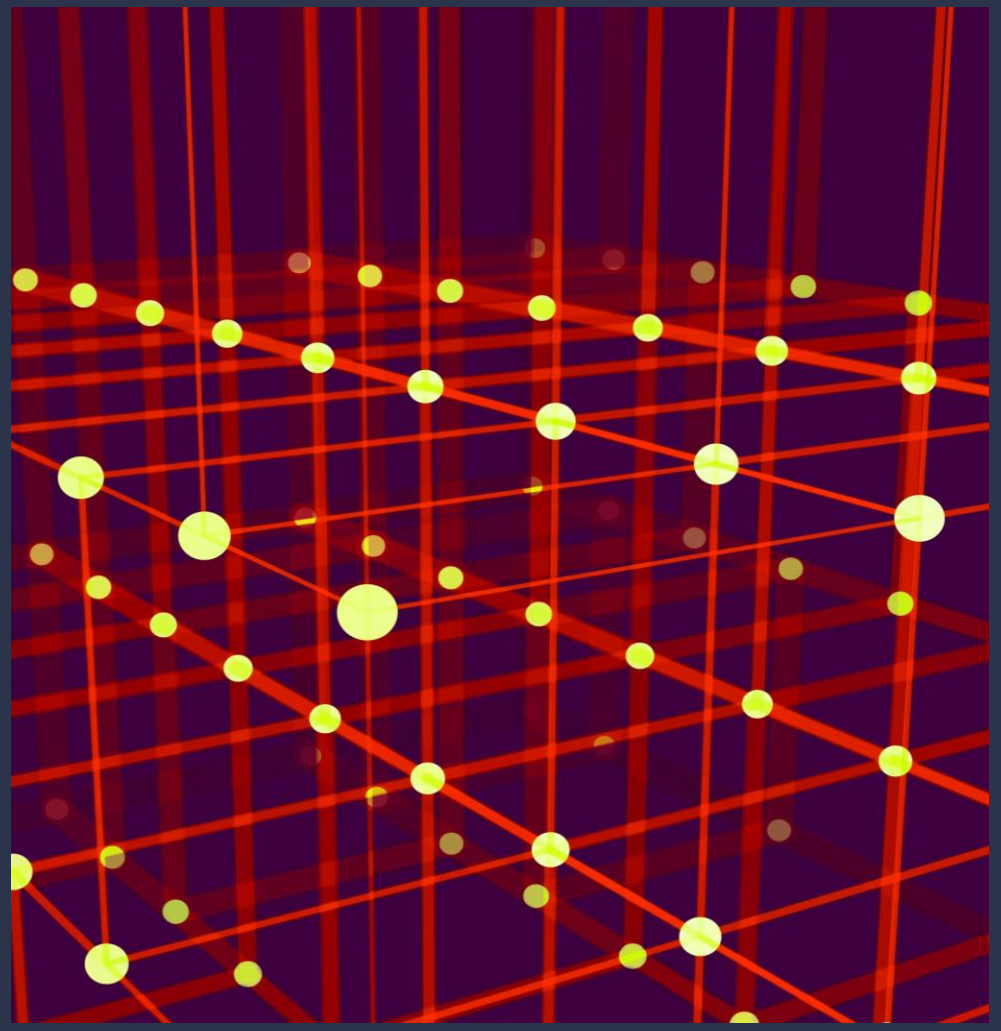
- Registres → L1/L2/L3 → RAM → NVMe → réseau
- Localité temporelle/spatiale
- Modèle Roofline : bande passante vs compute (op-intensity)

Modèles de programmation

- Mémoire partagée
  - Threads, données communes
  - Synchronisation (mutex, barrières, atomiques)
- Mémoire distribuée
  - Processus + messages (MPI)
  - Communicators, collectives
- Parallélisme
  - Data parallel (map/reduce)
  - Task parallel :
    - Pipeline : chaine traitement
    - Work stealing : vol de travail

Les différentes formes de parallélisme

- Parallélisme au niveau du bit (bit-level parallelism)
- Parallélisme au niveau de l'instruction (instruction-level parallelism)
- Parallélisme au niveau des données (data-level parallelism)
- Parallélisme au niveau des tâches (task-level parallelism)



Taxonomie de Flynn

- SISD (Single instruction on single data)
  - Aucun parallélisme
- SIMD (Single instruction on multiple data)
  - Parallélisme au niveau de la mémoire
- MISD (Multiple instructions on single data)
  - Donnée unique traitée par plusieurs unités de calcul en même temps. Peu utilisé en pratique
- MIMD (Multiple instructions on multiple data)
  - Plusieurs unités de calcul traitent des donnés différentes, chacune avec une mémoire distincte
  - (particulièrement utilisé sur les superordinateurs)

Single Instruction, Multiple Data

- SIMD décrit une classe d'instructions qui effectuent la même opération sur plusieurs registres simultanément.
- Exemple : Ajouter un scalaire à 3 registres, en stockant la sortie de chaque addition dans ces registres.
  - Utilisé pour augmenter la luminosité d'un pixel
- Les CPU disposent également d'instructions SIMD, qui sont très importantes pour les applications qui doivent effectuer beaucoup de calculs
  - Les codecs vidéo tels que x264/x265 utilisent largement les instructions SIMD pour accélérer l'encodage et le décodage vidéo.

Vers le SIMD...

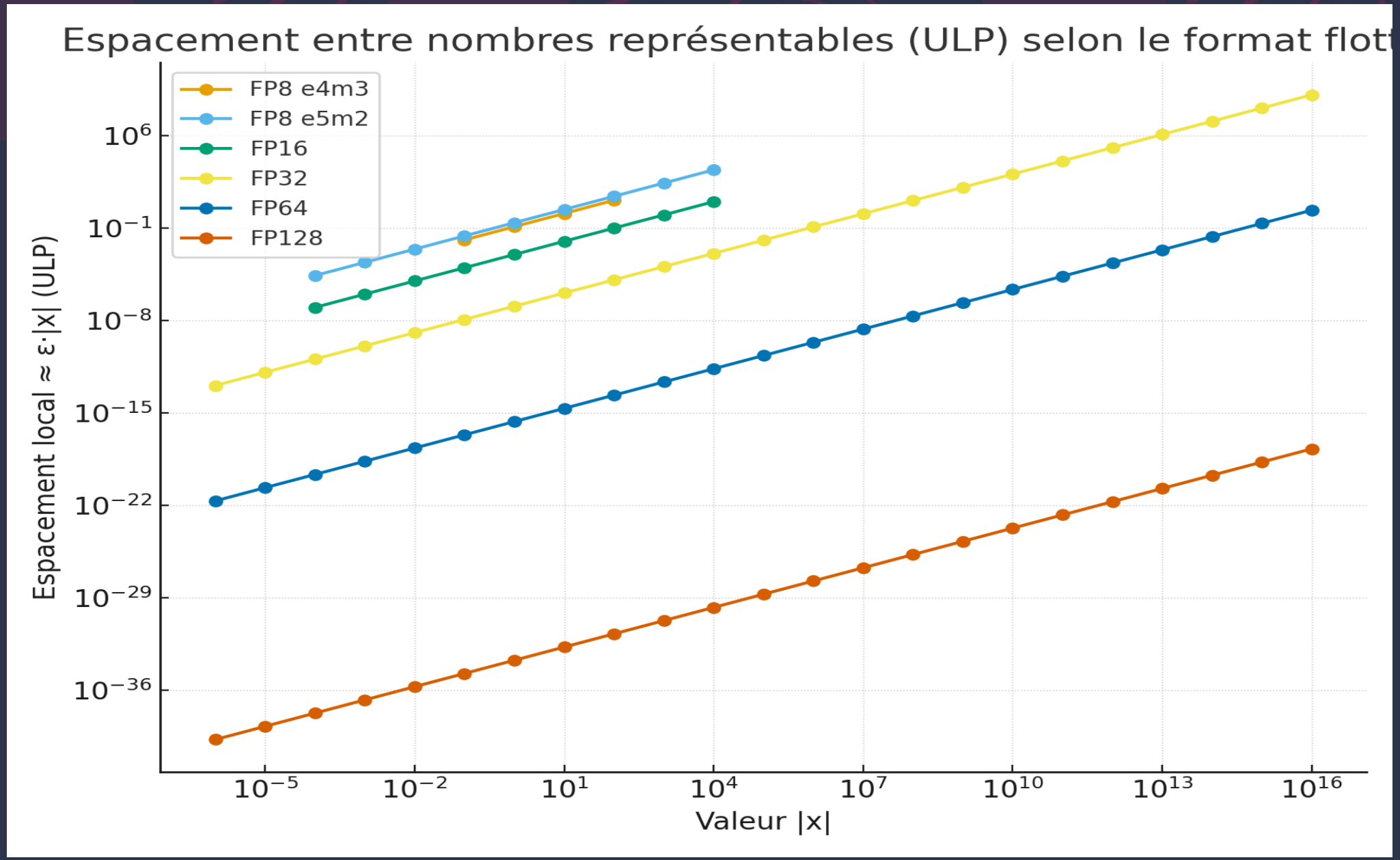
- La conversion d'un algorithme pour utiliser SIMD est généralement appelée « vectorisation »
- Attention : tous les algorithmes ne peuvent pas en bénéficier ou même être vectorisés
- L'utilisation des instructions SIMD n'est pas toujours bénéfique
  - Le SIMD nécessite une puissance supplémentaire, et donc un gaspillage de chaleur
  - Si les gains sont faibles, la complexité supplémentaire n'en vaut probablement pas la peine
- Plusieurs compilateurs sont développés actuellement afin de vectoriser eux-mêmes le code

Single Instruction, Multiple Thread

- Une extension plus souple de SIMD, qui est le modèle de calcul que CUDA utilise
- Principales différences :
  - Une seule instruction, plusieurs jeux de registre
  - Une seule instruction, plusieurs adresses (c.-à-d. accès parallèle à la mémoire !)
- Plusieurs compilateurs sont développés actuellement afin de vectoriser eux-mêmes le code
- Une seule instruction, plusieurs chemins d'accès

Calcul avec quelle précision ? Hardware 8, 16, 32, 64 bits

Format	Bits	Exposant/ Mantisse	Précision	Plage de valeur	Remarques
FP8 e4m3	8	4/3	**~1**	$1.6\times10^{-2} \rightarrow 2.4\times10^{2}$	Très faible précision, utile pour IA quantifiée (poids)
FP8 e5em2	8	5/2	**~1**	$6\times10^{-5} \rightarrow 5.7\times10^{4}$	Grande plage mais erreur énorme
FP16	16	5/10	**~3**	$6\times10^{-5} \rightarrow 6.55\times10^{4}$	Bon compromis pour IA et graphisme
FP32	32	8/23	**~7-8**	$1.17\times10^{-38} \rightarrow 3.4\times10^{38}$	Format standard GPU/CPU
FP64	64	11/52	**~15-17**	$4.94\times10^{-324} \rightarrow 1.8\times10^{308}$	Référence scientifique, très fiable
FPA128	128	15/112	**~34**	$3.36\times10^{-4944} \rightarrow 1.19\times10^{4932}$	Très haute précision (physique, calcul symbolique)


Espacement entre nombres représentables (ULP) selon le format flott
FP8 e4m3
FP8 e5m2
FP16
FP32
FP64
FP128
Espacement local ≈ ε·|x| (ULP)
10^6
10^-1
10^-8
10^-15
10^-22
10^-29
10^-36
10^-5
10^-2
10^1
10^4
10^7
10^10
10^13
10^16
Valeur |x|

Loi d’Amdahl

donne l'accélération théorique en latence de l'exécution d'une tâche *à charge d'exécution constante* que l'on peut attendre d'un système dont on améliore les ressources.

- Comprendre la limite théorique du parallélisme.
- Même avec des centaines de processeurs, le gain reste limité.

Loi d’Amdahl - Idée de base

- Un programme contient :
- - Une partie séquentielle (S)
- - Une partie parallélisable (P)

- Même avec beaucoup de processeurs, la partie séquentielle limite le gain global.

Loi d'Amdahl - Formule

- Speedup(N) = 1 / (S + P/N)

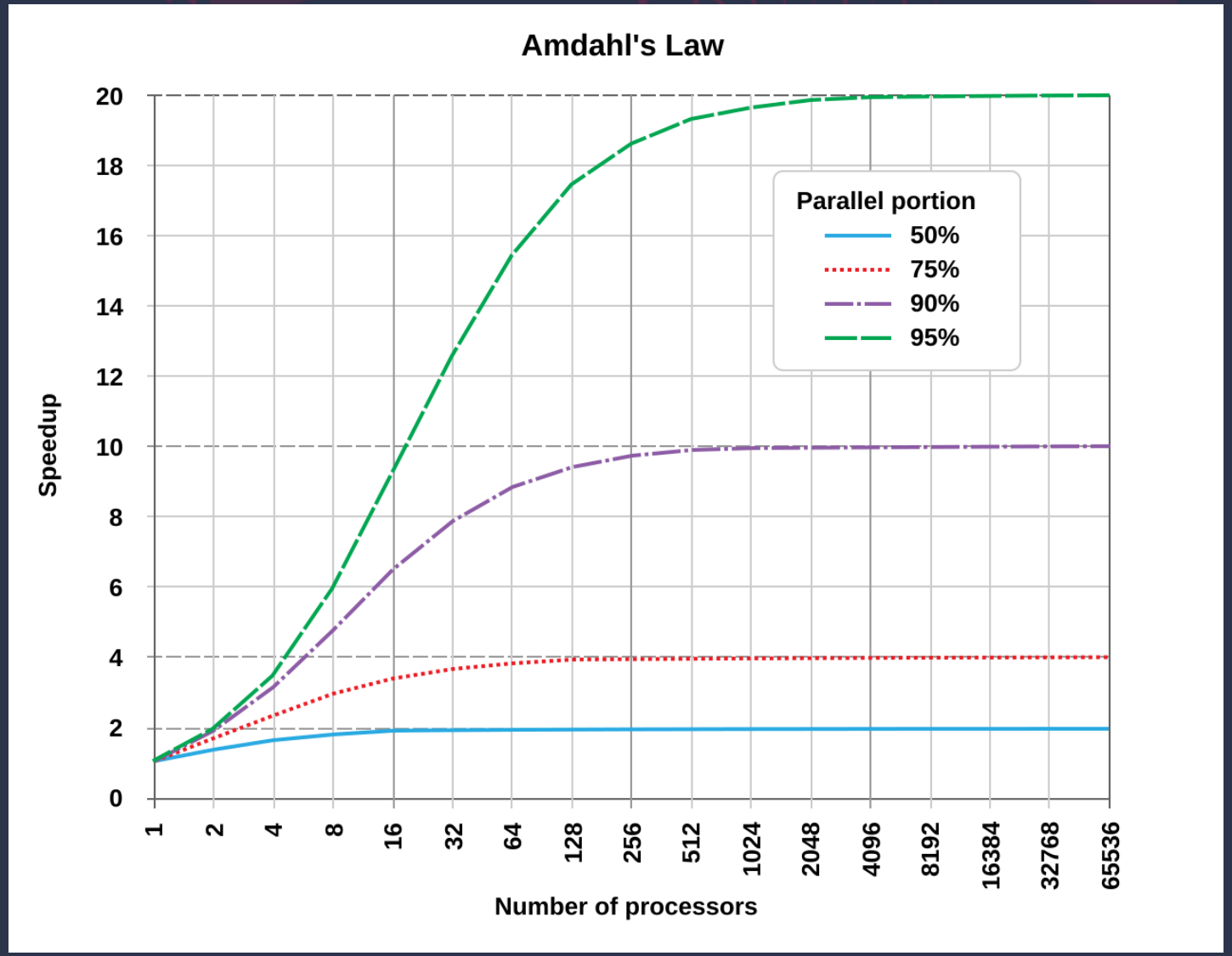
- N : nombre de processeurs
- S : partie séquentielle
- P : 1 - S : partie parallélisable

Loi d'Amdahl - Exemple numérique

- Programme avec 80 % parallélisable (P=0.8, S=0.2)

- N=1 → 1.0x
- N=2 → 1.67x
- N=4 → 2.5x
- N=8 → 3.33x
- N=100 → 4.76x

Loi d'Amdhal - Interprétation visuelle



Loi d’Amdahl - Intuition

- Chaîne de montage :
- - 80 % du travail réparti entre plusieurs ouvriers.
- - 20 % fait par une seule personne (contrôle final).

- → Le contrôle devient le goulot d'étranglement.

Loi d’Amdahl - Conséquences pratiques

- - Réduire la partie séquentielle est plus rentable que d'ajouter des processeurs.
- - Le parallélisme parfait n’existe pas.
- - Le gain réel dépend du code, du matériel, et de la communication.

Loi de Gustafson

donne l'accélération théorique en latence de l'exécution d'une tache à temps d'exécution constant.

- Comprendre comment le parallélisme peut continuer à offrir des gains lorsque la taille du problème augmente avec le nombre de processeurs.

Loi de Gustafson - Idée de base

- Contrairement à Amdahl, Gustafson suppose que la charge de travail augmente avec le nombre de processeurs.

- → On évalue le speedup en maintenant le temps constant, pas la taille du problème.

Loi de Gustafson - Formule

- Speedup(N) = N - S × (N - 1)

- où :
- N = nombre de processeurs
- S = fraction séquentielle du programme

- → Le gain est quasi linéaire si S est petit.

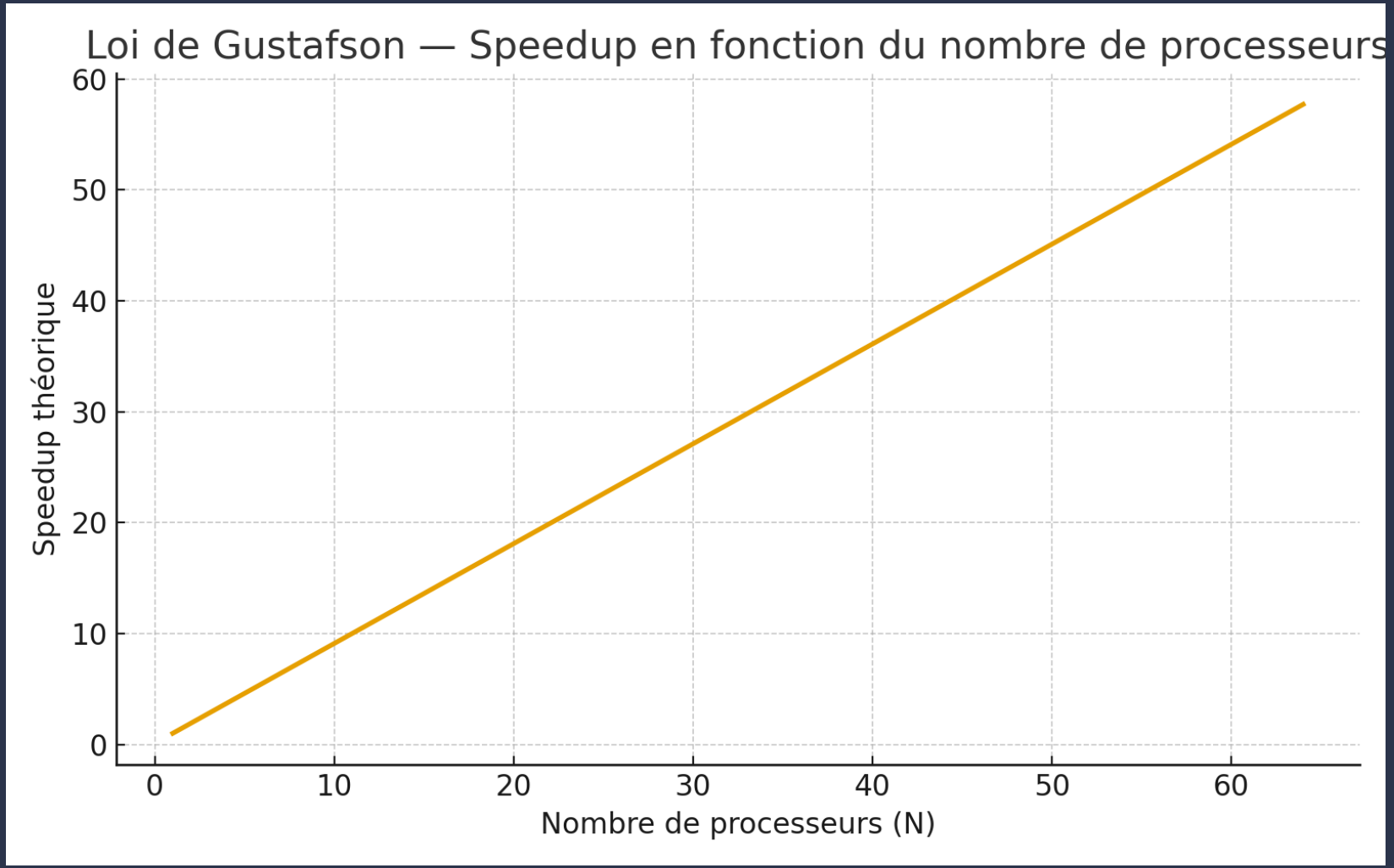
Loi de Gustafson - Exemple numérique

- Programme avec S = 0.1 (10 % séquentiel)

- N=1 → 1.0x
- N=4 → 3.7x
- N=8 → 7.3x
- N=100 → 90.1x

- ➡️ Le gain augmente presque linéairement avec N !

Loi de Gustafson - Interprétation visuelle



Loi de Gustafson - Intuition

- Quand on dispose de plus de processeurs, on ne cherche pas à exécuter le même problème plus vite, mais à résoudre un problème **plus grand** dans le même temps.

- → On exploite la puissance supplémentaire pour augmenter la taille du calcul.

Loi de Gustafson - Conséquences pratiques

- - Modèle plus réaliste pour les supercalculateurs.
- - Permet de mieux représenter le scaling des applications lourdes (simulation, rendu 3D, IA...).
- - Encourage à adapter la taille du problème au nombre de cœurs disponibles.

Comparaison Loi d’Amdahl vs Gustafson

- Amdahl : taille du problème **fixe** → speedup limité.
- Gustafson : taille du problème **variable** → speedup quasi linéaire.

- Amdahl → pessimiste pour le calcul parallèle.
- Gustafson → optimiste, plus représentatif en pratique.

Loi d’Amdahl vs loi de Gustafson

- - Amdahl : montre la **limite** du parallélisme.
- - Gustafson : montre le **potentiel** du parallélisme.

- Formules :
- Amdahl → 1 / (S + (1 - S)/N)
- Gustafson → N - S × (N - 1)

- 🎯 Moralité : Réduire S reste la clé, mais adapter la charge permet de mieux exploiter le parallélisme.

Le CPU (ou Central Processing Unit)

- Cœur de l'ordinateur, le CPU est un outil de calcul versatile
- Optimisé pour les taches séquentielles et la réduction de la latence
- Technologie fonctionnelle
- Conçu pour exécuter une seule tâche rapidement
- Généralement équipés de 8 cœurs/files, très puissants avec un contrôle complexe
- Cache de grande taille pour réduire les temps d’accès à la mémoire
- Mal adapté au calcul parallèle

Le GPU (ou Graphics Processing Unit)

- Technologie relativement nouvelle faite pour le calcul parallèle
- A l'origine créée pour résoudre des problèmes de rendu graphique
- Devenue de plus en plus capable de traiter de larges problèmes
- Très rapide et puissant
- Coçu pour le débit et pour l’exécution de nombreuses tâches simultanément
- Utilise beaucoup de puissance électrique
- Hiérarchie mémoire spécifique (global, shared, registers …)

Un exemple d'utilisation : le ray tracing

```
for all pixels (i,j) in image:
    From camera eye point,
      calculate ray point and
direction in 3d space
    if ray intersects object:
      calculate lighting at closest
object point
      store color of (i,j)
Assemble into image file
```

- Ressources en ligne : le site *shadertoy*

CPU vs GPU

- CPU:
  - Bon pour les tâches séquentielles, le contrôle complexe, les opérations avec peu de données.
  - Mauvais pour les tâches massivement parallèles, les opérations sur de grandes quantités de données.
- GPU:
  - Bon pour les tâches massivement parallèles, les opérations sur de grandes quantités de données.
  - Moins bon pour les tâches séquentielles, le contrôle complexe, les opérations avec peu de données.

Un exemple simple : l'addition

- Ajouter deux vecteurs A et B afin d'obtenir un vecteur C

A[] + B[] -> C[]

- Sur le CPU:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
C[i] = A[i] + B[i];
return C;
```

- Sur CPU le code ci-dessous s'exécute séquentiellement. Peut-on faire mieux ?

Un exemple simple : l'addition

- Sur CPU (multi-threaded, pseudocode) :

```
(allocate memory for array C)
Create # of threads equal to number of cores on processor (around 2, 4, perhaps 8?)
(Indicate portions of arrays A, B, C to each thread...)

...

In each thread,
For (i from beginning region of thread)
C[i] <- A[i] + B[i]
//lots of waiting involved for memory reads, writes, ...
Wait for threads to synchronize...
```

  - Ce code est plus rapide (de 2 à 8 fois...)

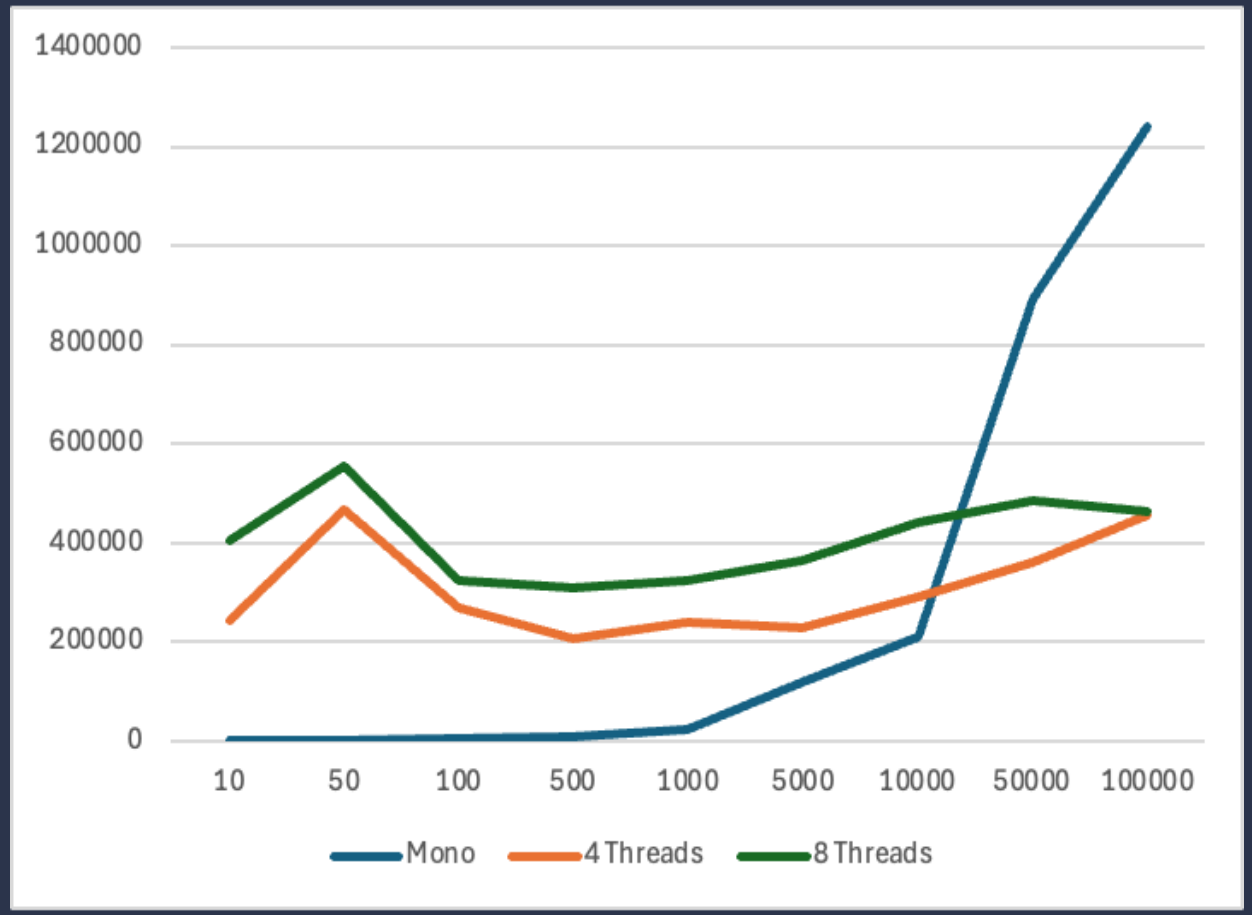
Un exemple simple : l'addition

- Combien de threads sont accessibles sur le CPU ? Comment la performance évolue-t-elle avec le nombre de threads ?
  - (En pratique, chaque CPU contient deux threads).

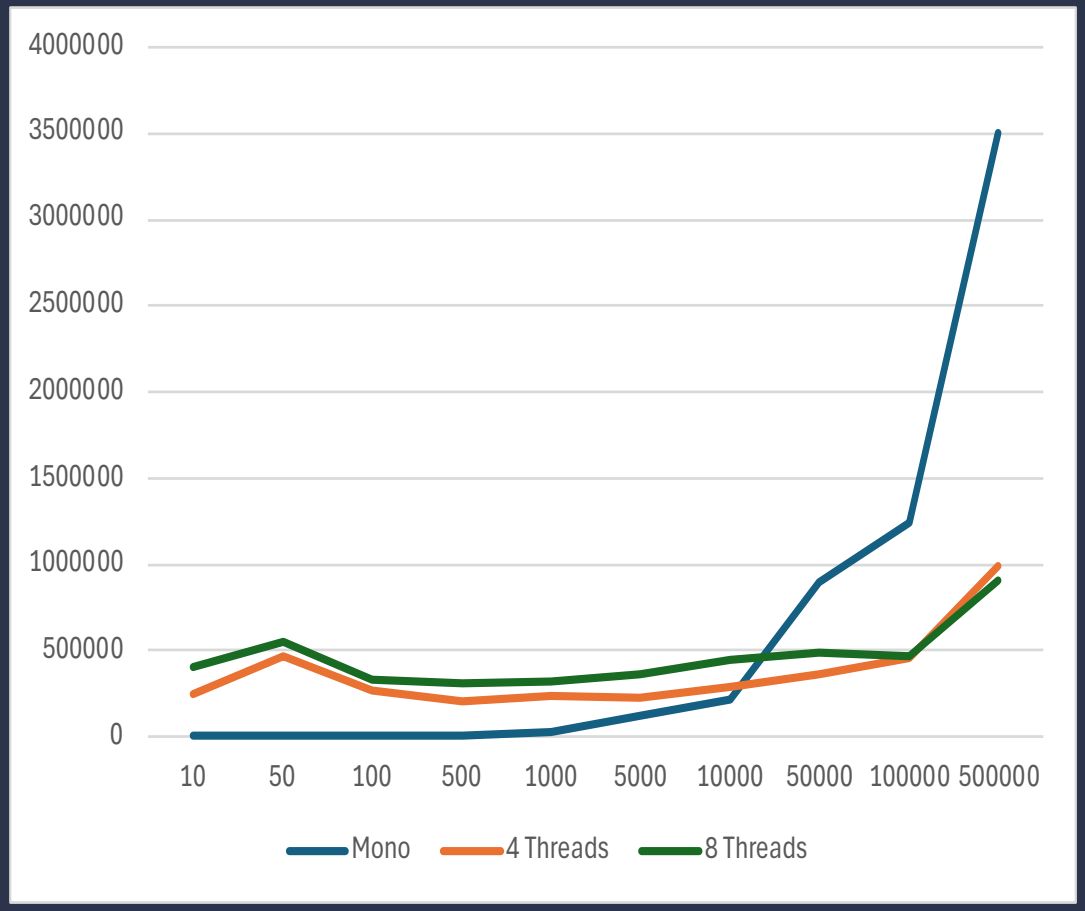
- Context switching :
  - L'action de changer de thread exécutant le code
  - **Forte pénalité pour le CPU**
  - Pas de pénalité pour le GPU

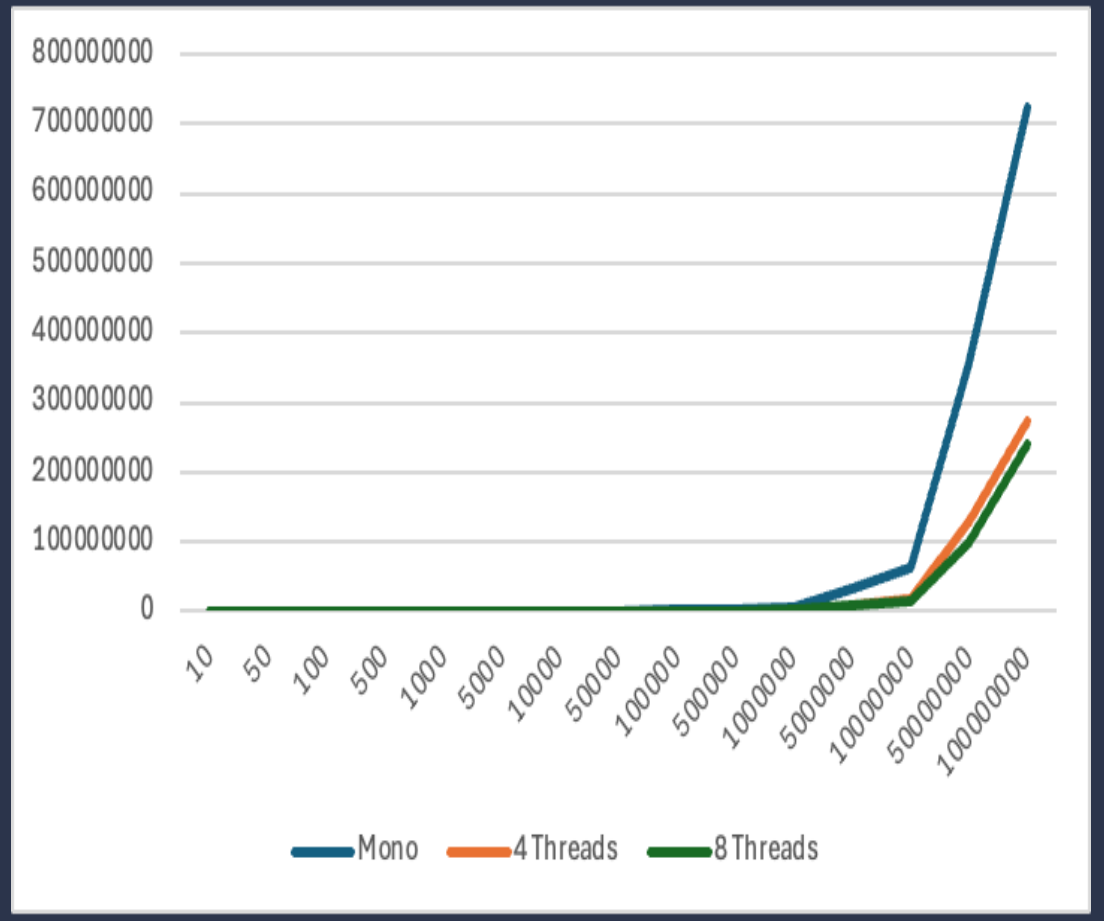
Addition de vecteurs sur CPU (Mac M2)

Taille	Temps en ns		
	Mono	4 Threads	8 Threads
10	375	244167	406166
50	2417	468042	553875
100	4750	267833	325791
500	8917	207666	308459
1000	23292	238792	323167
5000	118125	228917	364084
10000	212041	289791	443459
50000	892833	361042	485667
100000	1239459	457042	462833
500000	3502416	987625	903167
1000000	6065541	1780083	1647833
5000000	32228542	8710459	7820333
10000000	63548458	17204458	15518833
50000000	357172833	129256917	99675833
100000000	724879334	271749416	238867042



Addition de vecteurs sur CPU (Mac M2)





Un exemple simple : l'addition

- Sur GPU (multi-threaded, pseudocode) :

```
(allocate memory for arrays A, B, C on GPU)
Create the “kernel” – where each thread will perform one (or a few) additions
Specify the following kernel operation:

For all i‘s (indices) assigned to this thread:
C[i] <- A[i] + B[i]
```

  - Peut démarrer 20 000 threads (!) à la fois !

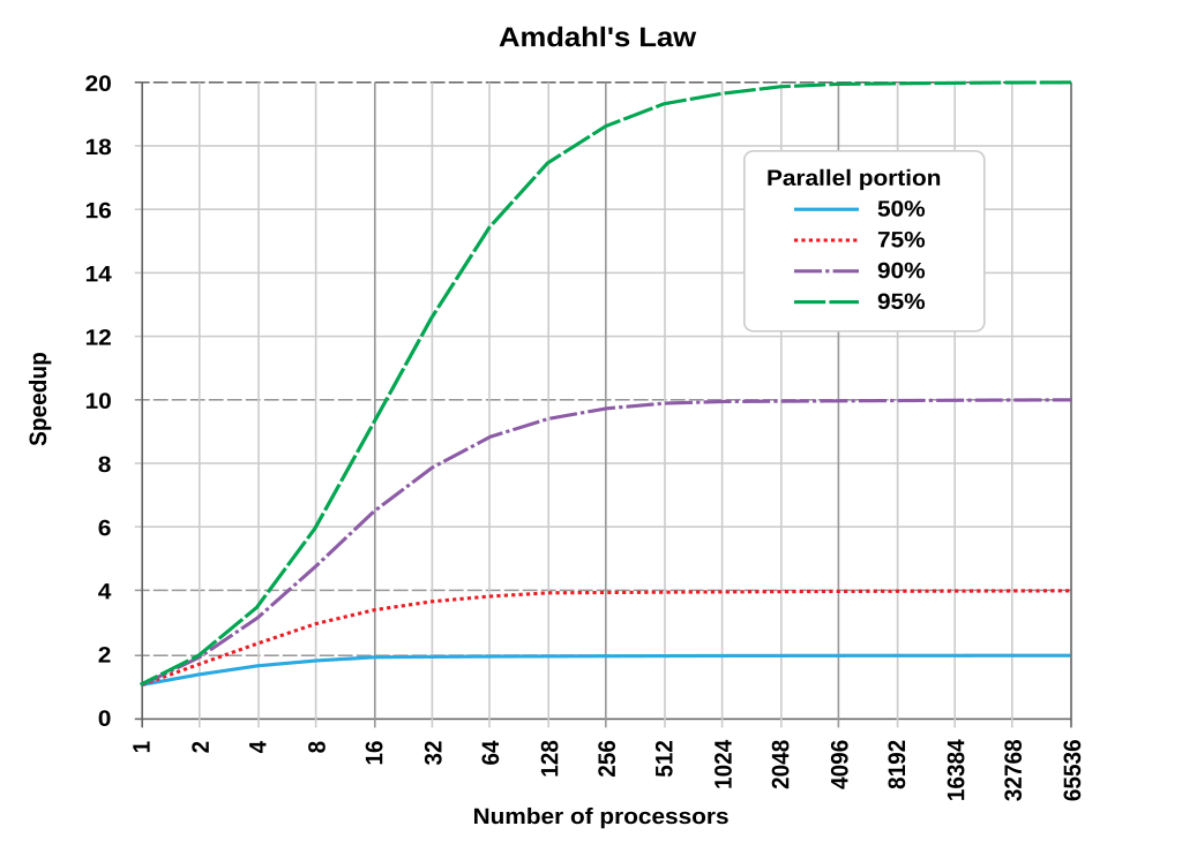
Les avantages du GPU

- Le très fort parallélisme des GPU nous permet de profiter des nombreux coeurs de la machine
- Cela nous permet d'exécuter simultanément de très nombreux threads

Plus de threads...?

- La loi d'Amdahl

L'accélération théorique est limitée par la partie sérielle du programme. Par exemple, si 95 % du programme peut être parallélisée, l'accélération théorique maximale en utilisant le calcul parallèle est de 20 fois.



GPU : Une utilisation graphique

- Les shaders
  - Les routines graphiques permettent d'implémenter ses propres fonctions à l'aide de nombreux langages.
  - GLSL OpenGL Shading Language
  - Vulkan Programmation graphique
  - OpenCL Open Computing Language
  - ...

GPU : Une utilisation graphique

- General-purpose computing on GPUs (GPGPU)
  - Le matériel est devenu suffisamment performant pour que l'on puisse disposer d'un mini-superordinateur.
- CUDA (Compute Unified Device Architecture)
  - Plate-forme de calcul parallèle polyvalente pour les GPU NVIDIA
- Vulkan/OpenCL (Open Computing Language)
  - Frameworks hétérogènes très puissants
- Les deux sont accessibles à partir de différents languages
  - Pour Python, il y a Theano, pyCUDA...
- D'autres langages de programmation se préparent au passage GPU : Julia(open source)

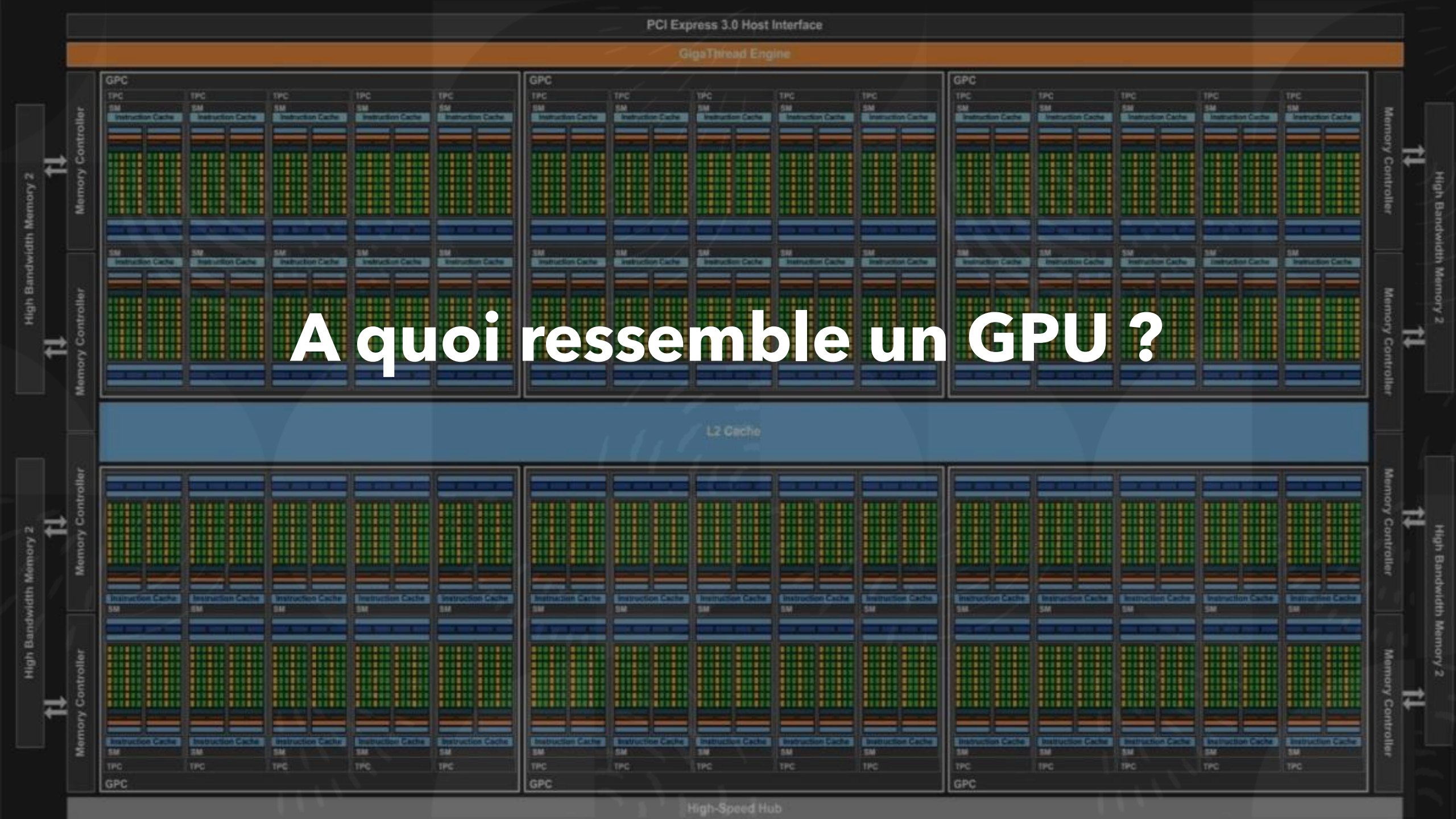
https://julialang.org/

GPU : Un champ d'application plus large

- Résolution d'équations aux dérivées partielles
- Résolution de problèmes de mécanique des fluides
- Résolution par méthodes Monte Carlo
- Deep learning
- Traitement du signal sur GPU
- ...

Un calcul GPU étape par étape

- Configurer les entrées sur l'hôte (mémoire accessible à l'unité centrale)
- Allouer de la mémoire pour les sorties sur le CPU de l'hôte
- Allouer de la mémoire pour les entrées sur le GPU
- Allouer de la mémoire pour les sorties sur le GPU
- Copie des entrées de l'hôte vers le GPU (lente)
- Démarrer le noyau du GPU (fonction qui s'exécute sur le GPU - rapide !)
- Copier les sorties du GPU vers l'hôte (lent)

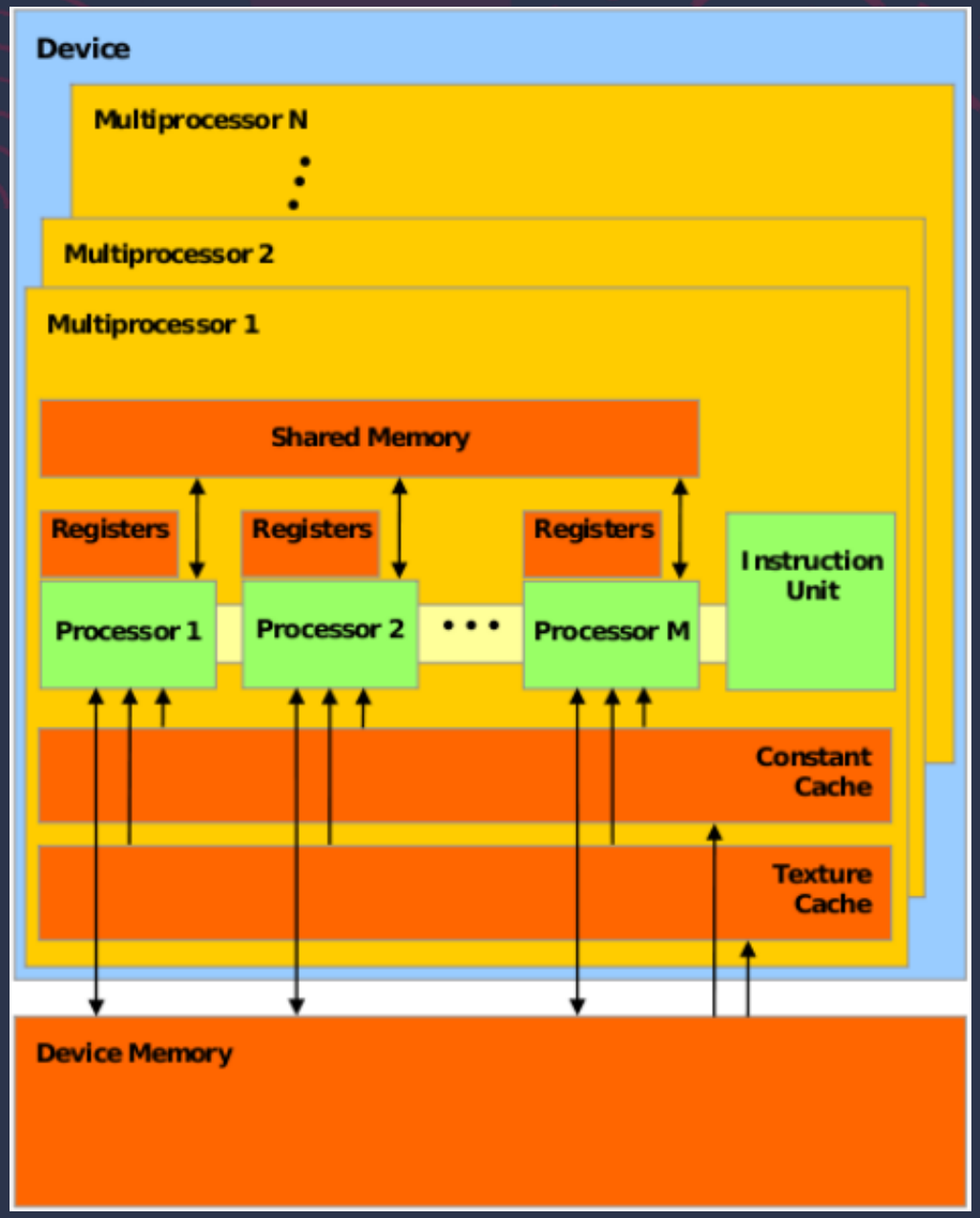


A quoi ressemble un GPU ?


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub

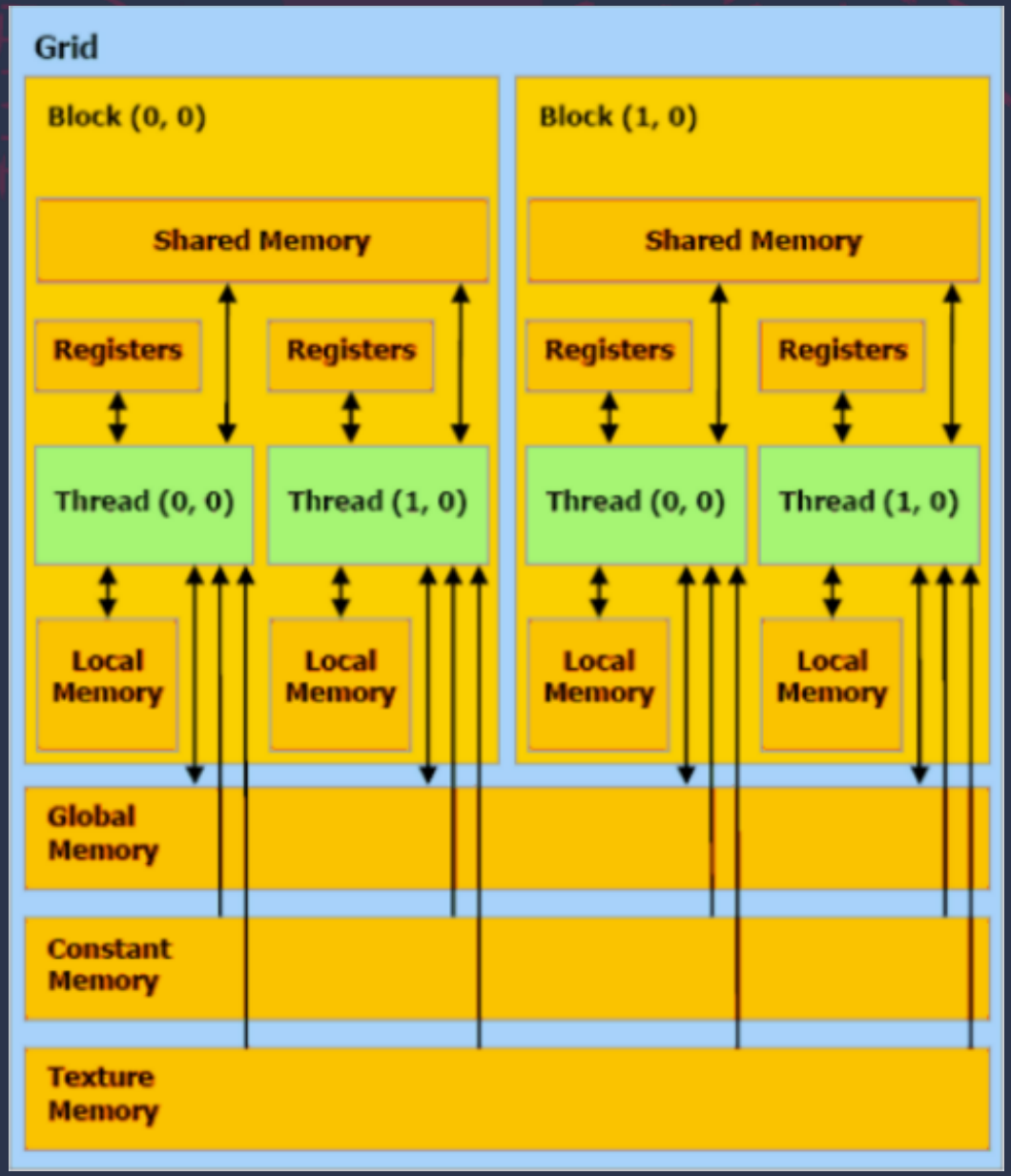
A l'intérieur d'un GPU...

- Device Memory (ou Global Memory) correspond à la RAM du GPU
  - Plus rapide qu'accéder à la RAM de la machine elle-même
- Les GPU ont de nombreux Streaming Multiprocessors (SMs)
  - Chaque SM possède plusieurs processeurs mais une seule unité d'instruction
  - Les groupes de processeurs doivent exécuter exactement le même jeu d'instructions à tout moment à un moment donné dans un seul SM



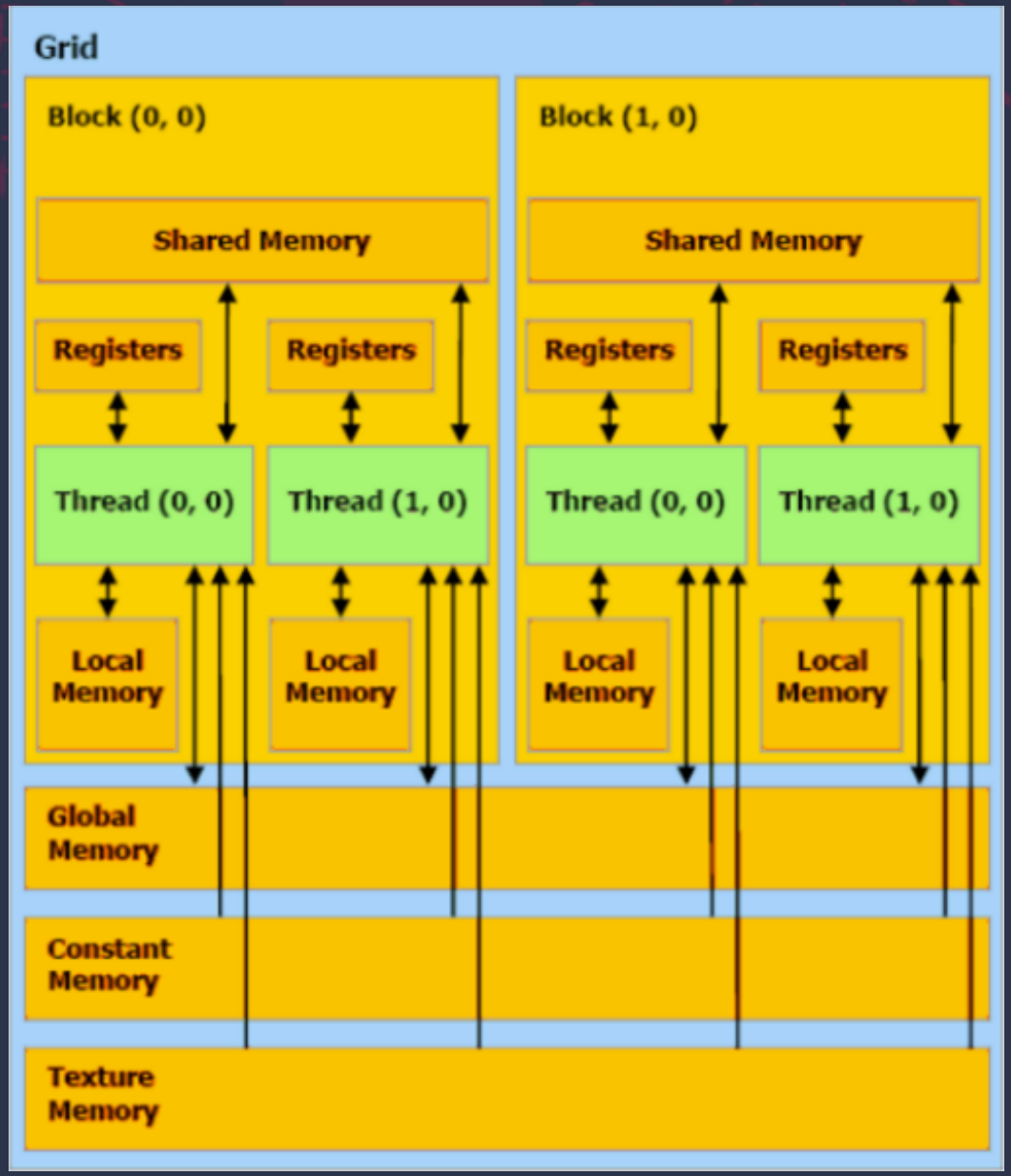
Organisation thread-block

- Thread
  unité de calcul la plus petite sur le GPU, capable d'exécuter des instructions indépendamment des autres threads.
- Block
  groupe de threads qui exécutent le même kernel. Peuvent communiquer entre eux grâce à la mémoire partagée.
- Grid
  grid est l'ensemble des blocs lancés pour exécuter un kernel sur le GPU.



La mémoire GPU en détail

- Registres
- Mémoire locale
- Mémoire globale
- Mémoire partagée
- Etc...



La mémoire GPU en détail

- Registres
  forme de mémoire la plus rapide sur le multiprocesseur. N'est accessible que par le thread. A la durée de vie du thread lui-même.

- Mémoire locale
  Réside dans la mémoire globale et peut être 150 fois plus lente que la mémoire de registre ou la mémoire partagée. Elle n'est uniquement accessible par le thread. A la durée de vie du thread.

La mémoire GPU en détail

- Mémoire globale
  Peut être aussi rapide qu'un registre lorsqu'il n'y a pas de conflit de banque ou lorsque la lecture se fait à partir de la même adresse. Accessible par n'importe quel thread du bloc à partir duquel il a été créé. A la durée de vie du bloc.

- Mémoire partagée
  Potentiellement 150x plus lente que le registre ou la mémoire partagée. Accessible à partir de l'hôte. A la durée de vie de l'application, c'est-à-dire qu'elle est persistante entre les lancements du noyau.

La mémoire globale

- Physiquement séparée du coeur du GPU !
- La très vaste majorité de la mémoire du GPU est de la mémoire globale
- Si les données d'entrée sont trop importantes, il faudra les diviser avant de les envoyer
- Les GPU ont aujourd'hui de 0.5 à 25 Go de mémoire globale, la majorité en ayant au moins 2 Go.
- L'entrée-sortie de la mémoire globale est très lente !
- La plus lente du tout le GPU (en dehors de l'accès à l'hôte)
- Elle doit être faite aussi peu que possible
- Latence : ~ 300 ns

La mémoire partagée (Shared Memory)

- Mémoire très rapide stockée dans le SM (Streaming Multiprocessor)
- N'est pas accessible en dehors d'un bloc
- Taille de l'ordre de... 48 ko !
- Peut être allouée dynamiquement lors de l'exécution, ou statiquement (de taille connue dès la compilation)

- Latence : ~ 5 ns

- Différents outils de programmation GPU

- C++
- Non-propriétaire

- C++ / Swift
- Propriétaire

- C
- Non-propriétaire

- C++ / Swift
- Propriétaire