

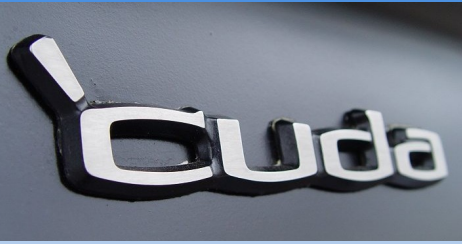
Programmation sur GPU avec CUDA

Pierre Kunzli

hpc, hepia ISC, 2025

Généralités: GPGPU

- GPU (*Graphics Processing Unit*) = chip massivement multi-thread, multi-coeur
 - Plusieurs milliers de threads concurrents assignés aux processeurs du GPU (centaines de processeurs scalaires exécutant tous le même programme séquentiel)
 - Parallélisme de données à grain fin
 - Grands tableaux de données, flux important
- GPGPU = *General Purpose computation using GPU* pour des applications autres que du 3D graphique



Généralités: CUDA

- CUDA = Compute Unified Device Architecture
 - Architecture de calcul parallèle développée par NVIDIA
 - Moteur de calcul des GPU de NVIDIA
 - Accessibilité dans un langage de programmation standard, des instructions et de la mémoire des GPUs pour du calcul parallèle
 - Modèle de programmation général permettant le lancement de paquets de threads sur un GPU
- Driver pour charger des programmes de calcul sur un GPU
 - Driver autonome, optimisé pour le calcul
 - Interface dévolue au calcul, API libre d'aspects graphiques
 - Partage de données avec OpenGL via des objets tampons
 - Gestion explicite de la mémoire du GPU



Généralités: la librairie CUDA

- Ensemble minimal d'extensions au langage C
 - Programmation de portions de code source à destination du GPU
 - API + librairies standard en C
 - CUBLAS : implémentation de BLAS (algèbre linéaire)
 - CUFFT : implémentation de FFT
- CUDA se scinde en composants s'exécutant soit sur CPU, soit sur GPU

Modèle de programmation

- GPU (=device) sert de coprocesseur de calcul au CPU (=host)
 - possède sa propre DRAM (device memory aussi appelé GRAM)
 - exécute plusieurs threads en parallèle indépendamment du CPU
- Parties data-parallèle d'une application s'exécute sur le device en tant que kernels
 - Un kernel exécute plusieurs threads concurrents

Modèle de programmation

- Différences entre threads GPU et threads CPU
 - Threads GPU sont extrêmement légers
 - Très peu d'overhead à la création
 - Un GPU a besoin de milliers de threads pour être pleinement efficient
 - Un CPU multi-coeur n'a besoin que de peu de threads

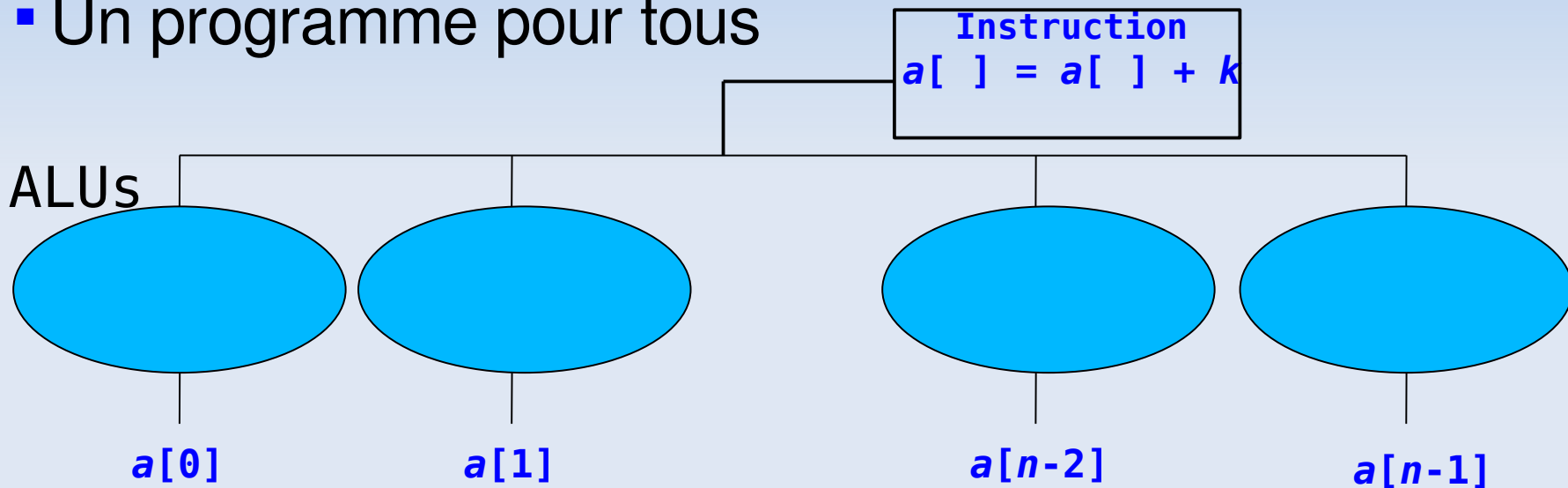
Modèle de programmation

- GPUs historiquement conçus pour calculer les données des images à afficher à l'écran
 - Manipulation des pixels de l'image en appliquant souvent la même opération à chaque pixel
- Modèle **SIMD/T (Single Instruction Multiple Data/Thread)**
 - Mode opératoire où la même opération est effectuée simultanément sur chaque donnée



Modèle de programmation SIMD/T (Single Instruction Multiple Data/Thread)

- SIMD = parallélisme de données
- Une instruction spécifie l'opération à faire
- Un programme pour tous



- SIMT = version SIMD pour GPUs
- Plusieurs threads exécutant chacun la même séquence d'instructions

Applications adaptées au modèle SIMT

- Opérations matricielles
 - Même opérations sur les éléments de matrice
- Méthodes de Monte Carlo parallèles
 - Utilisation d'un grand nombre de tirages aléatoires indépendants
- Manipulations de données
 - Algorithmes de tri

Différents types d'applications CUDA

DIVERSE WORKLOADS IN MODERN CLOUD COMPUTING



Graphics



Scientific Computing



Data Analytics



**AI Deep Learning
Training**



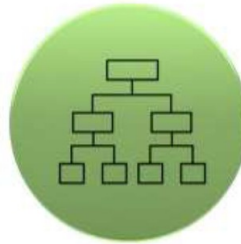
**Edge AI Video
Analytics**



Cloud Gaming



Genomics



**Classical Machine
Learning**



**AI Deep Learning
Inference**



5G Private Networks

Architecture hardware CUDA

- Composants primaires de l'architecture GPU Nvidia
 - Streaming Multiprocessor (SM)
 - p.ex. 20 SM avec 128 Cuda Cores sur la GeForce GTX 1080
 - Total de 2560 coeurs CUDA
 - Cuda Core (Anciennement Scalar Processor)
 - Hiérarchie mémoire
 - Réseau d'interconnexion
 - Interface avec le host

Streaming Multiprocessor (SM)

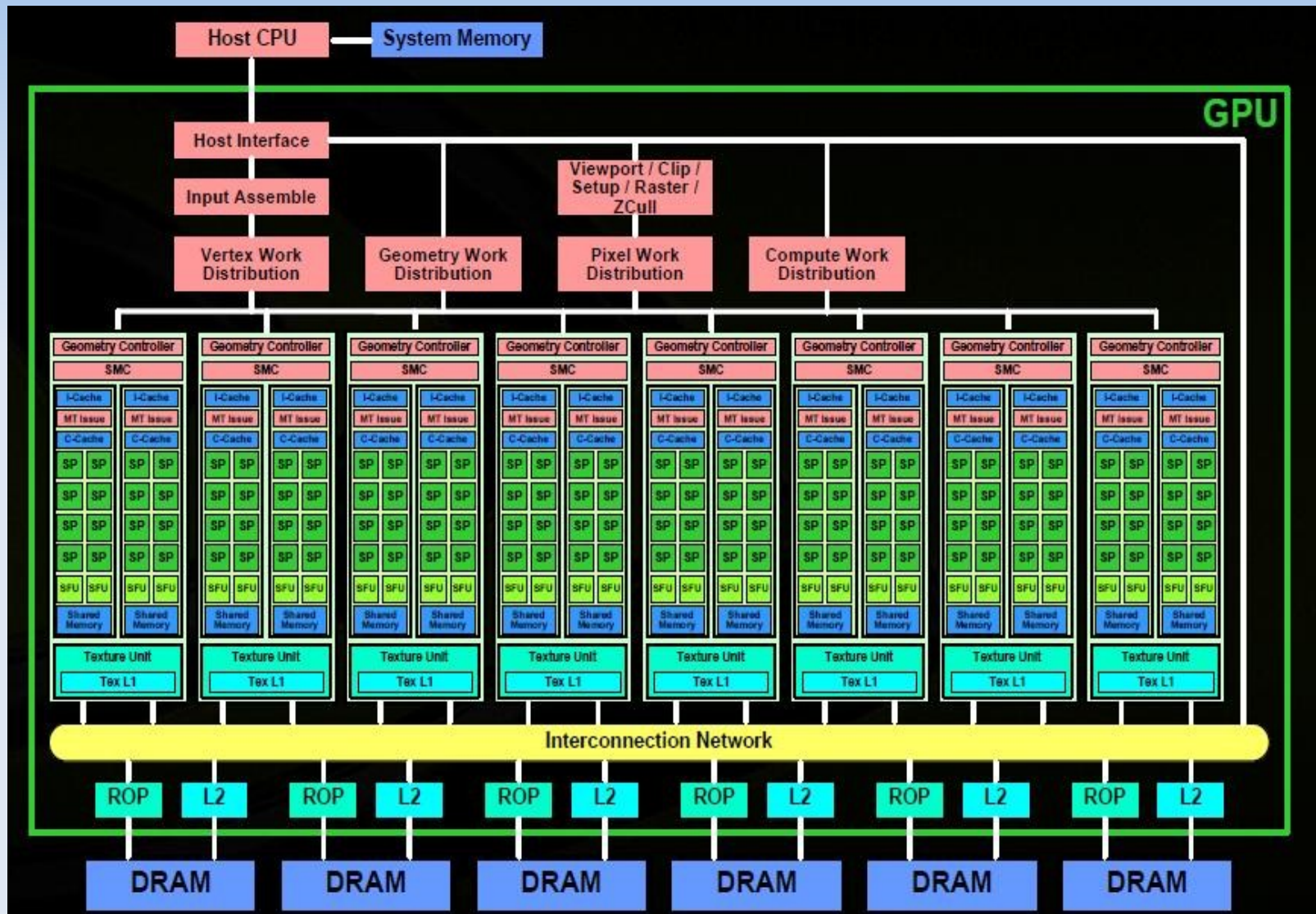
Scalar Processor (SP)



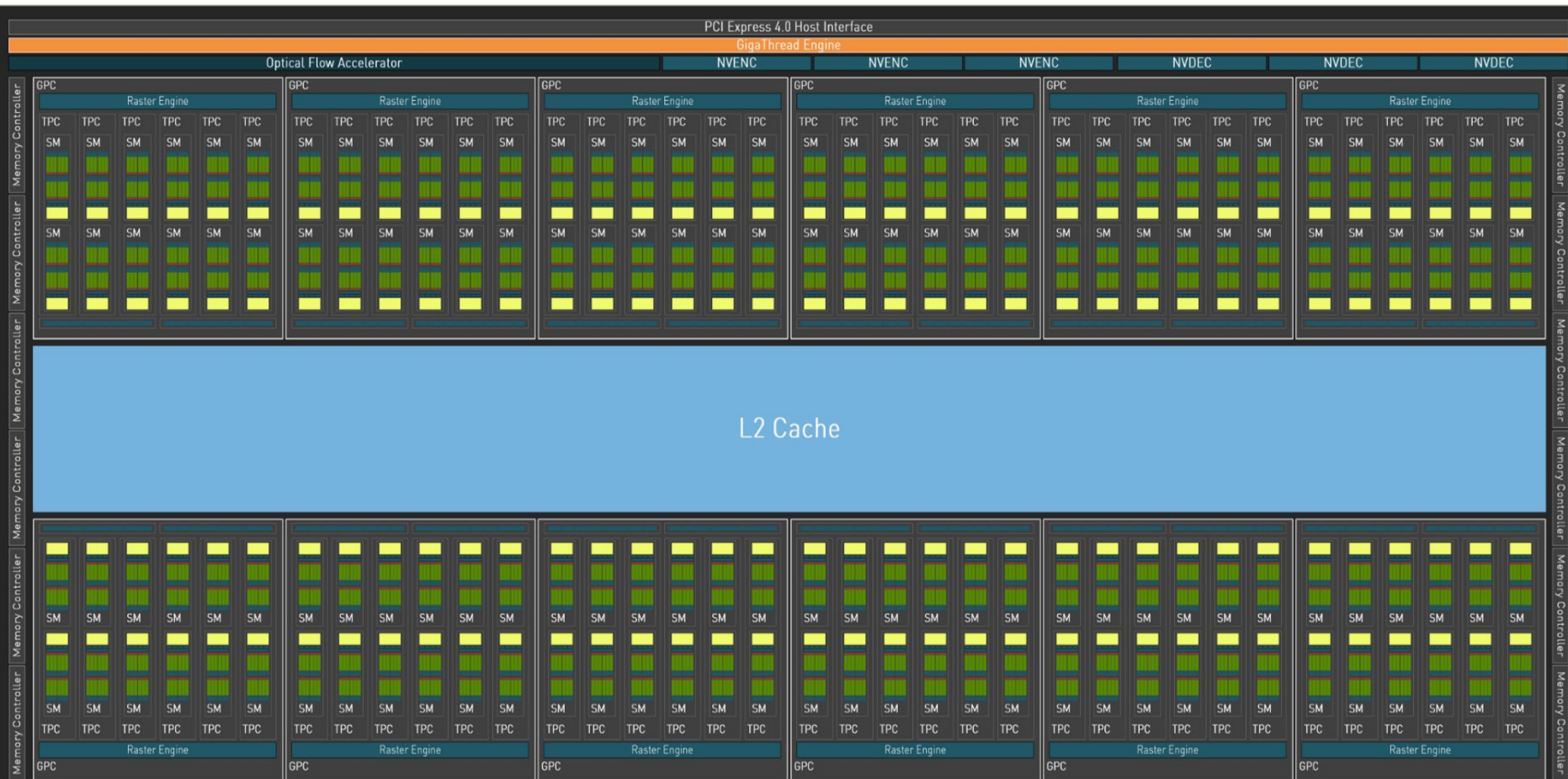
■ Architecture Tesla (2006)

- Chaque SM est constitué de 8 SP
- Chaque SP
 - supporte des instructions en virgule flottante FADD, FMAD, FMIN, FMAX, FSET, F2I, I et des opérations entières 32-bit : IADD, IMUL24, IMAD24, SHL, AND, OR, XOR,
 - pleinement pipeliné
 - cadencé à 1.35 GHz (32 Gflops en pointe)
 - dispose de 8'192 registres 32-bit dynamiquement alloués
 - supporte matériellement 768 threads (24 groupes SIMT de 32 threads)
 - ordonnancement matériel des threads
 - 16KB de mémoire partagée à faible latence
 - 2 Special Function Units (racine carrée, fonctions trigonométriques, ...)

Architecture matérielle d'un GPU



Architecture Ada, 2023, gaming



Note: The AD102 GPU also includes 288 FP64 Cores (2 per SM) which are not depicted in the above diagram. The FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations. The small number of FP64 Cores are included to ensure any programs with FP64 code operate correctly, including FP64 Tensor Core code.

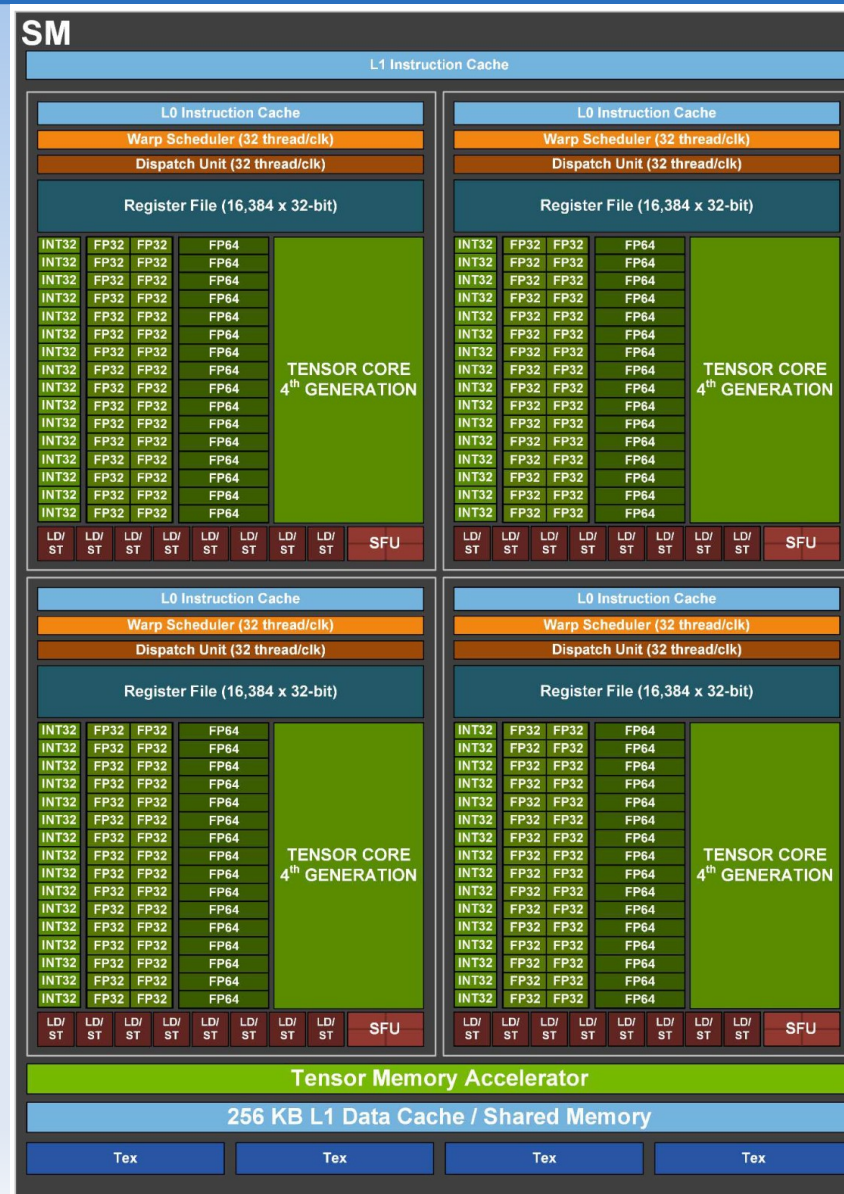
Architecture Ada, 2023, gaming



Architecture Hopper, 2023, datacenter



Architecture Hopper, 2023, datacenter



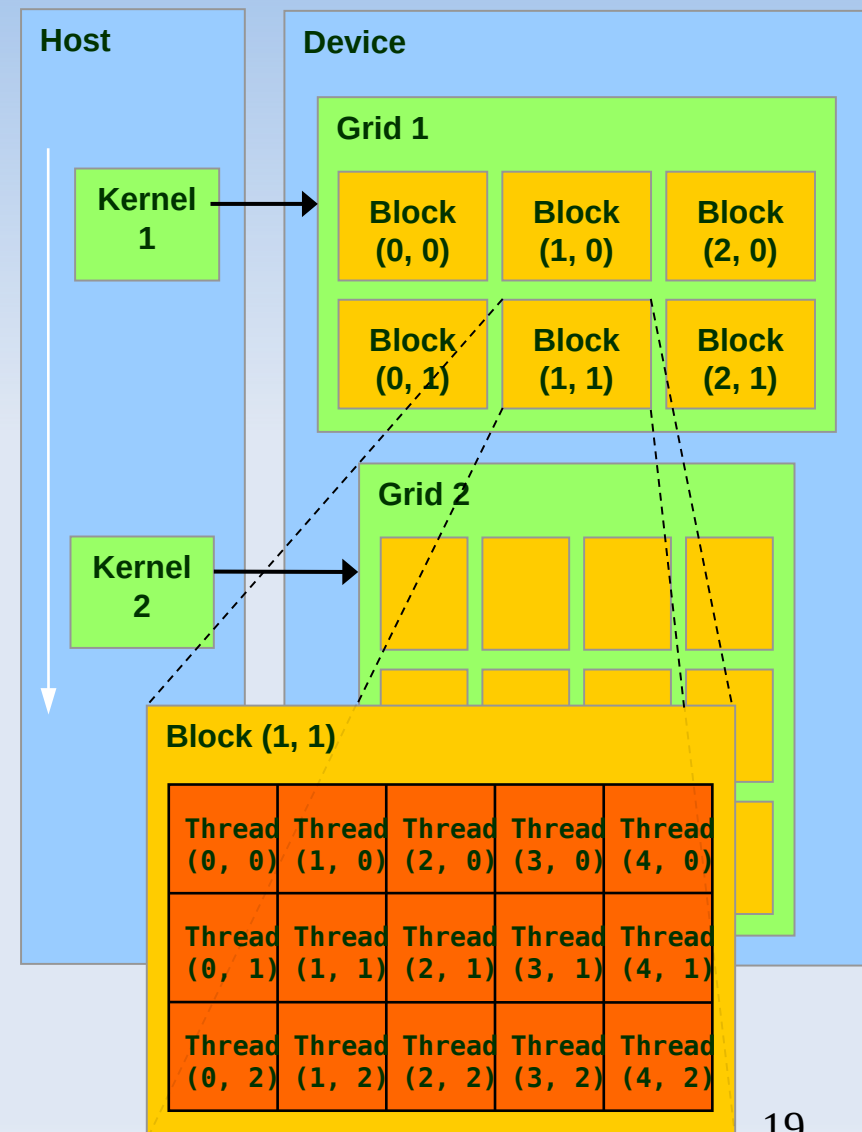
Evolution des architectures CUDA

- Retour de la spécialisation du matériel
 - Apparition des Tensor cores et Raytracing cores
 - Suppression des unités FP64 des modèles «gaming»
 - Modèles «datacenter» sans Raytracing cores

Modèle de programmation CUDA

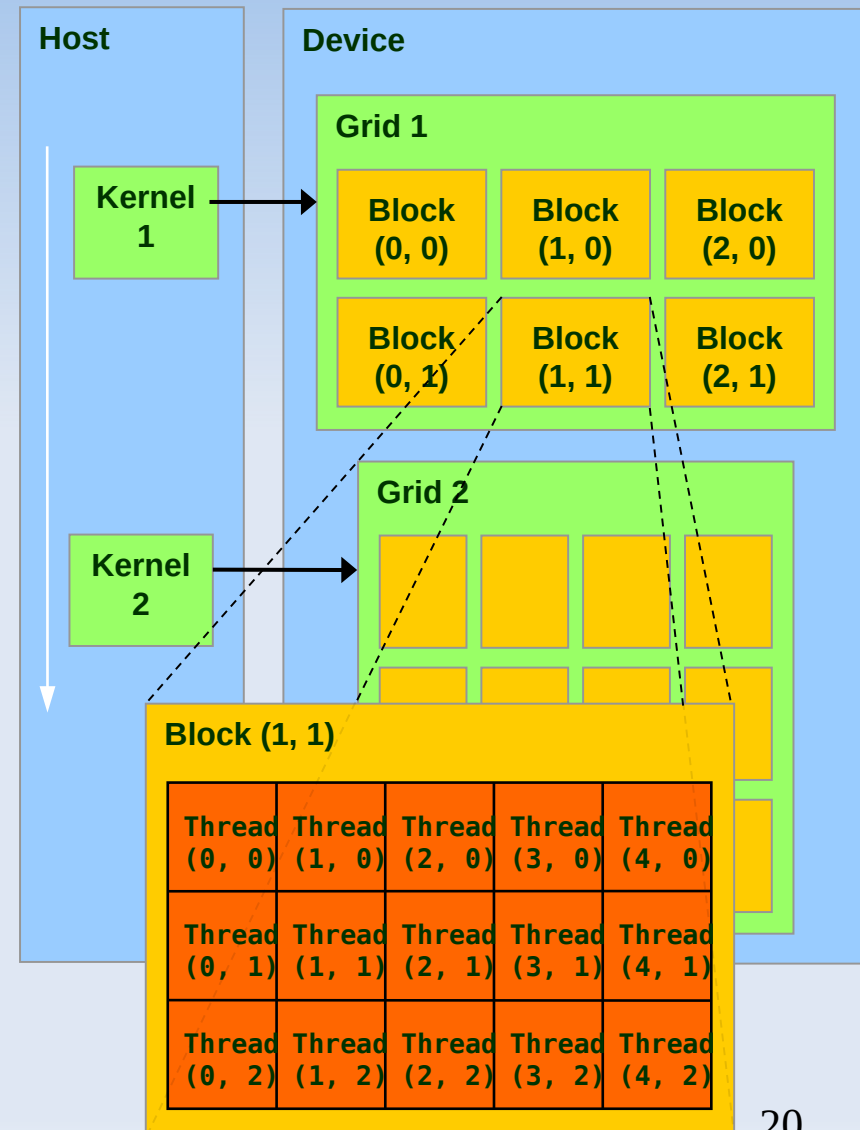
- Threads groupés en *blocks*
- *Blocks* groupés en un *grid*
- Un *grid* est exécuté sur le GPU comme *kernel*
- Un *grid* est créé pour chaque fonction kernel CUDA

Host = CPU
Device = GPU
Kernel = fonction s'exécutant sur le device



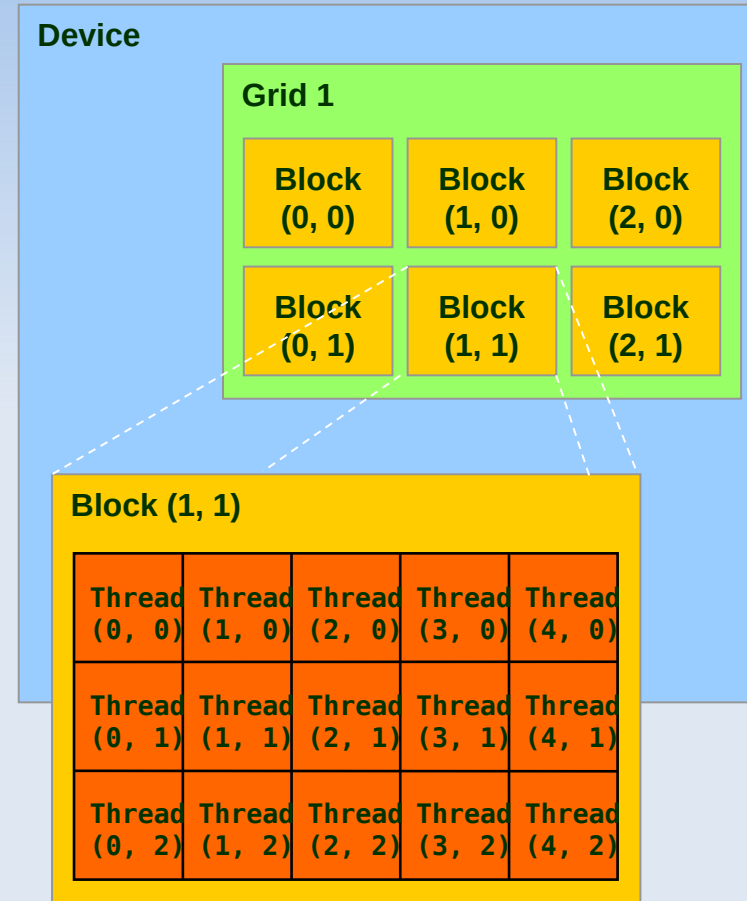
Groupes de threads: grids et blocks

- Un kernel est exécuté comme **grid de blocks de threads**
 - Tout thread peut accéder aux données en DRAM du GPU
- Un **block de threads** est un groupe de threads qui s'exécute sur un SM pouvant **coopérer** en
 - Synchronisant leur exécution
 - Partageant des données via une **mémoire partagée** à faible latence
- Deux threads dans des blocks différents ne peuvent pas coopérer « directement »
- Les threads d'un block sont exécutés par groupes nommés «warps»



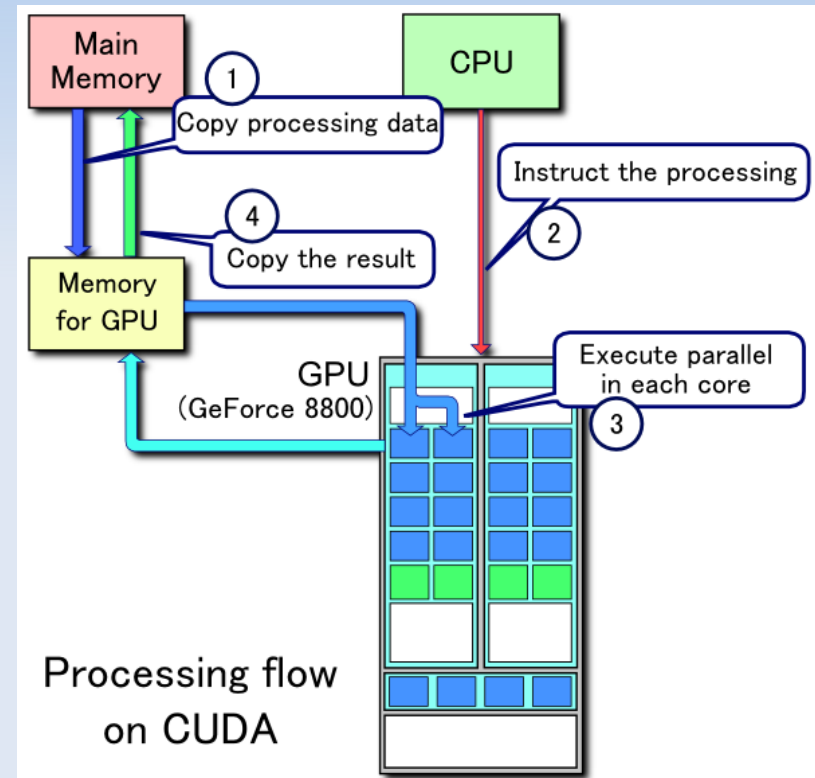
IDs des blocks et threads

- Threads et blocks ont des IDs
 - Chaque thread peut ainsi décider quelles données traiter
 - Block ID: 1D, 2D ou 3D
 - Thread ID: 1D, 2D ou 3D
- Les IDs simplifient l'adressage mémoire lors du traitement de données multidimensionnelles
 - Traitement d'image
 - Résolution d'équations différentielles sur des volumes



Flot de traitement CUDA

- (1) Copie des données de la RAM du CPU vers la GRAM du GPU
- (2) Le CPU charge les instructions de traitement sur le GPU
- (3) Le GPU exécute les instructions en parallèle sur chaque cœur
- (4) Copie des résultats de la GRAM du GPU vers la RAM du CPU

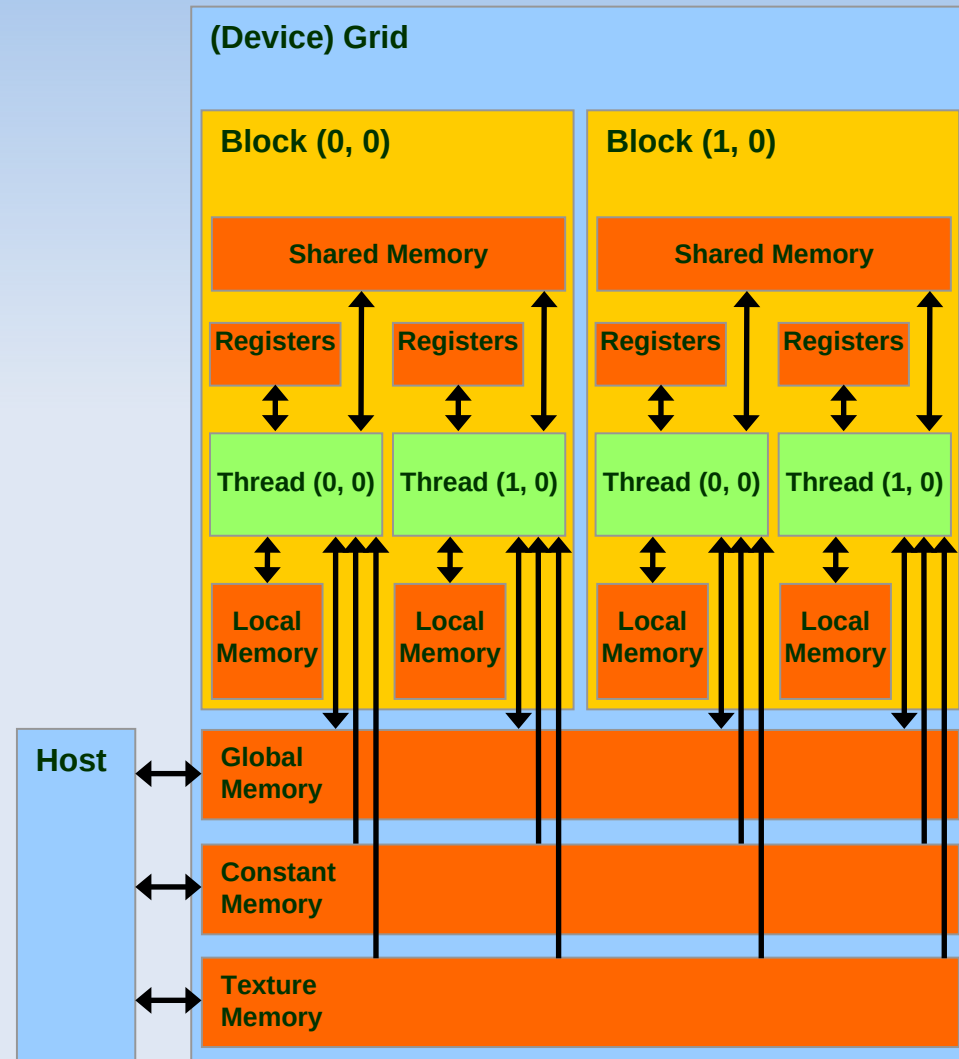


Hiérarchie mémoire CUDA

- Trois types principaux de mémoire
 - **Mémoire locale** – mémoire par thread pour variables locales et débordement de registres
 - **Mémoire partagée** – mémoire à faible latence par block pour partage de données et synchronisation intra-block, barrière de synchronisation via `__syncthreads()`
 - On-chip, très large bande passante, 100-150x plus rapide que la mémoire globale
 - **Mémoire globale (GRAM ou DRAM)** – mémoire au niveau du device partageable entre les blocks ou grids

Hiérarchie mémoire CUDA

- Chaque thread peut
 - R/W per-thread **registers**
 - R/W per-thread **local memory**
 - R/W per-block **shared memory**
 - R/W per-grid **global memory**
 - Read only per-grid **constant memory**
 - Read only per-grid **texture memory**
- Le host peut R/W les mémoires globales, constante, et de texture
- La mémoire partagée est plus rapide que les mémoires locale et globale, et aussi rapide que les registres s'il n'y a pas de conflits d'accès
- Les mémoires globale, constante, et de texture sont accessibles au CPU et au GPU



Développement: idée de base

1. Allocation mémoire de même taille sur le host et sur le device
2. Transfert des données du host vers le device
3. Exécution du kernel qui traite les données
4. Transfert inverse des données du device vers le host

Appel d'une fonction kernel

- Une fonction kernel doit être appelée avec une configuration d'exécution

```
int blocksize = 16;  
int gridsize = 4;  
dim3 dimBlock(blocksize);  
dim3 dimGrid(gridsize);  
kernel_function<<<dimGrid, dimBlock>>>(params);
```

- Un appel à une fonction kernel est asynchrone
- Une synchronisation explicite est nécessaire pour un rendez-vous

Identificateurs de threads et blocks

Chaque block et thread ont un ID unique prédéfini dans un grid et dans un block

- **threadIdx** – identificateur d'un thread
- **blockIdx** – identificateur d'un block
- **blockDim** – taille d'un block

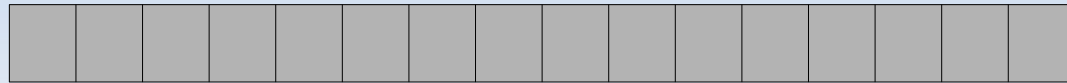
On peut définir un thread ID global unique

- **$\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$**

Exemple:

incrément des éléments d'un tableau

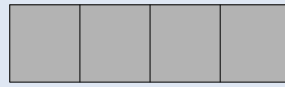
- Ajout d'une valeur **b** aux **N** éléments d'un tableau **a**
 - Exemple avec **N=16**, **blockDim=4**



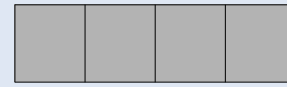
4 blocks



blockIdx.x=0
blockDim.x=4
threadIdx.x
=0,1,2,3
idx=0,1,2,3



blockIdx.x=1
blockDim.x=4
threadIdx.x
=0,1,2,3
idx=4,5,6,7



blockIdx.x=2
blockDim.x=4
threadIdx.x
=0,1,2,3
idx=8,9,10,11



blockIdx.x=3
blockDim.x=4
threadIdx.x
=0,1,2,3
idx=12,13,14,15

idx = blockIdx.x * blockDim.x + threadIdx.x
(correspondance entre index local et global)

Exemple: incrément des éléments d'un tableau, code device (kernel)

```
__global__  
void vadd(int *a, int *b, int n)  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if(i<n) a[i] += b[i];  
}
```

Exemple: incrément des éléments d'un tableau, code host

```
int main(){  
    ...  
    int *d_a, *d_b;  
    cudaMalloc((void**)&d_a, size);  
    cudaMalloc((void**)&d_b, size);  
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);  
    ...  
    dim3 dimBlock(blocksize);  
    dim3 dimGrid(gridsize);  
    vadd<<<dimGrid, dimBlock>>>(d_a, d_b);  
    cudaMemcpy(a, d_a, size, cudaMemcpyDeviceToHost);  
    cudaFree(d_a);  
    cudaFree(d_b);  
    ...  
}
```

Quelques remarques

- Une fonction kernel (global) ne doit rien retourner
- On crée en général plus de threads qu'il n'y a de données à traiter, il faut donc veiller à éviter les accès illégaux à la mémoire
- On peut afficher des données depuis un kernel avec printf, à n'utiliser que pour du debug

Synchronisation sur le CPU

Tout kernel est lancé de manière asynchrone

- Contrôle rendu immédiatement au CPU
- Un kernel commence son exécution dès que tout appel CUDA précédent s'est terminé

Les copies de mémoire sont synchrones

- Contrôle rendu au CPU dès que la copy est achevée
- La copie débute dès que tout appel CUDA précédent s'est terminé

Les appels asynchrones CUDA

- Permettent des copies en GRAM non-bloquantes
- Offrent la possibilité de superposer copies en GRAM et exécution de kernel
- L'API CUDA driver permet un accès à des fonctions bas niveau

Quelques liens

- Une introduction à CUDA en C++

<https://developer.nvidia.com/blog/even-easier-introduction-cuda/>

- NVIDIA CUDA Programming guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>