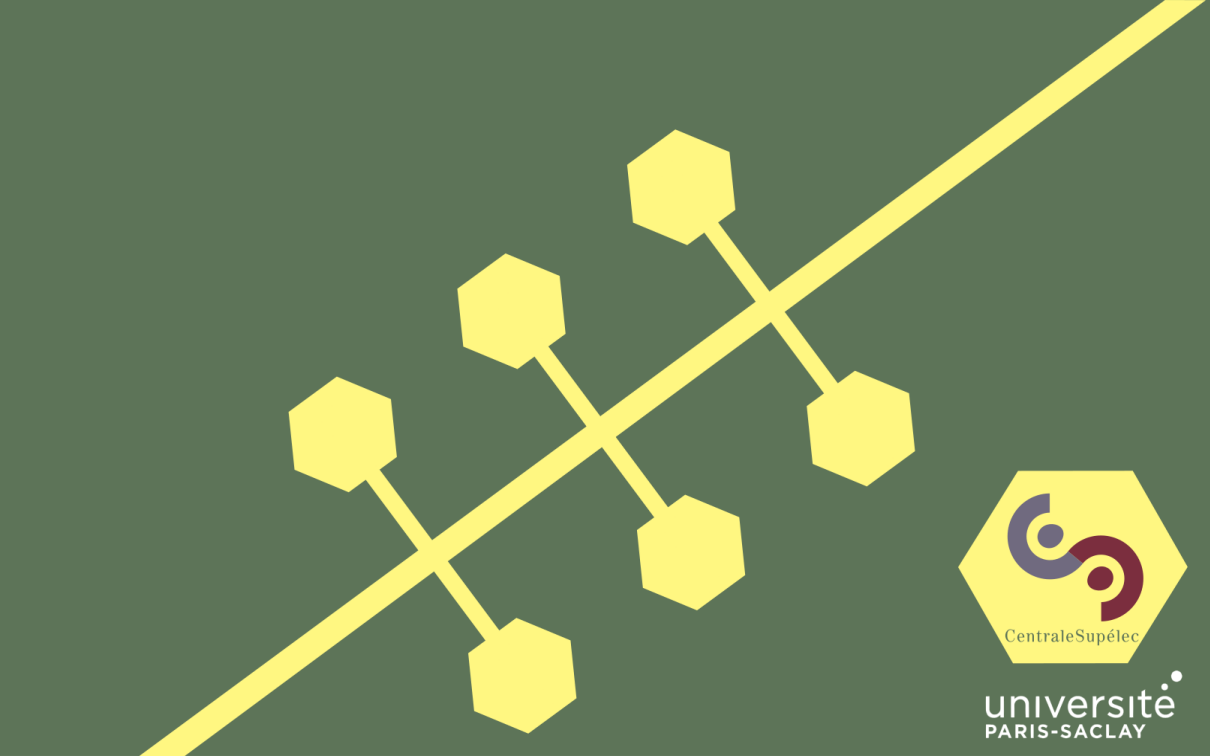
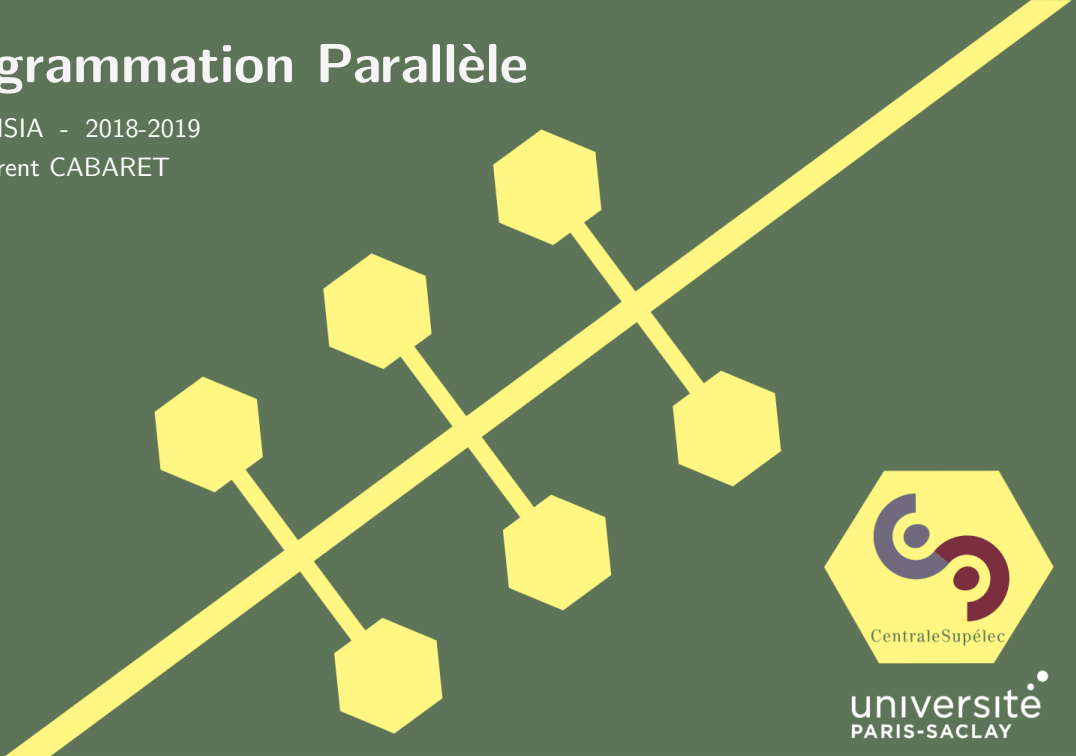

CentraleSupélec
université
PARIS-SACLAY

Programmation Parallèle

Option ISIA - 2018-2019

par Laurent CABARET



CentraleSupélec

université PARIS-SACLAY

Plan

- **Présentation** : Objectifs du cours, déroulement, évaluation, illustrations
- **Ab uno disce omnes** : Architecture séquentielle et mur de la chaleur
- **Paradigme parallèle**
- **Parallélisme d'instruction** : Vectorisation et SIMD
- **Parallélisme multi-cœur** : Mémoire partagée, OpenMP, Race conditions
- **Parallélisme massif** : Cas du GPU, programmation CUDA

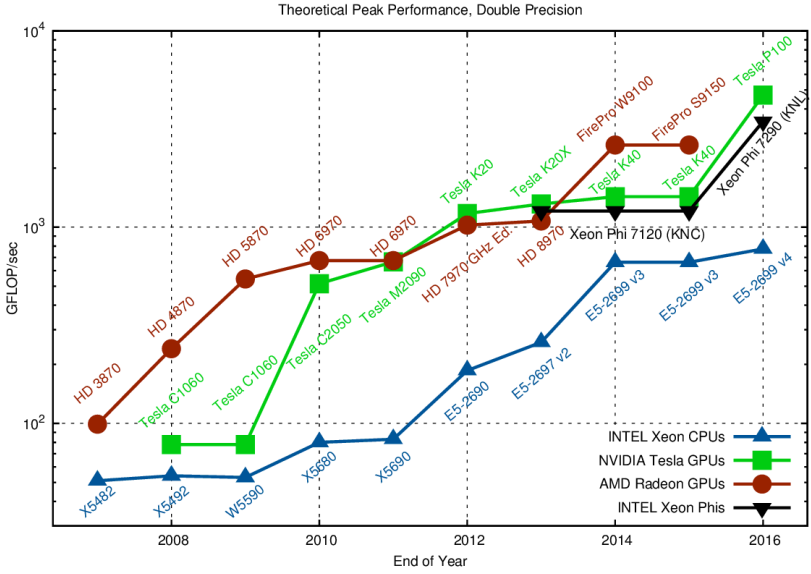
Parallélisme massif avec CUDA

Intéret du parallélisme massif par l'exemple

Version longue - https://www.youtube.com/watch?v=ZrJeYFxpUyQ

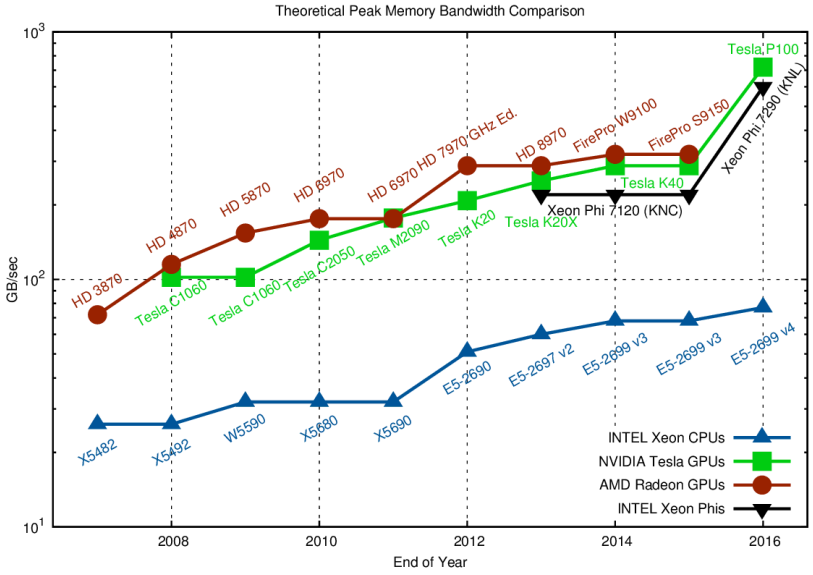
Spoiler : les défauts du GPU sont aussi visibles dans la vidéo !

Intéret du parallélisme massif : Flops ...



P100 : 5.3 TFLOPS - V100 : 7.8 TFLOPS

Intéret du parallélisme massif : ... et bandwidth

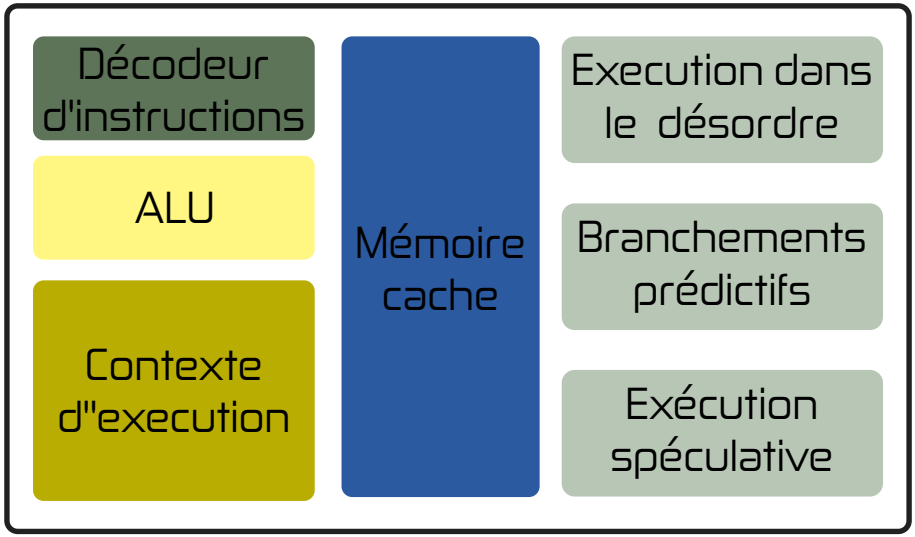


P100 : 732 GB/s - V100 : 900 GB/s

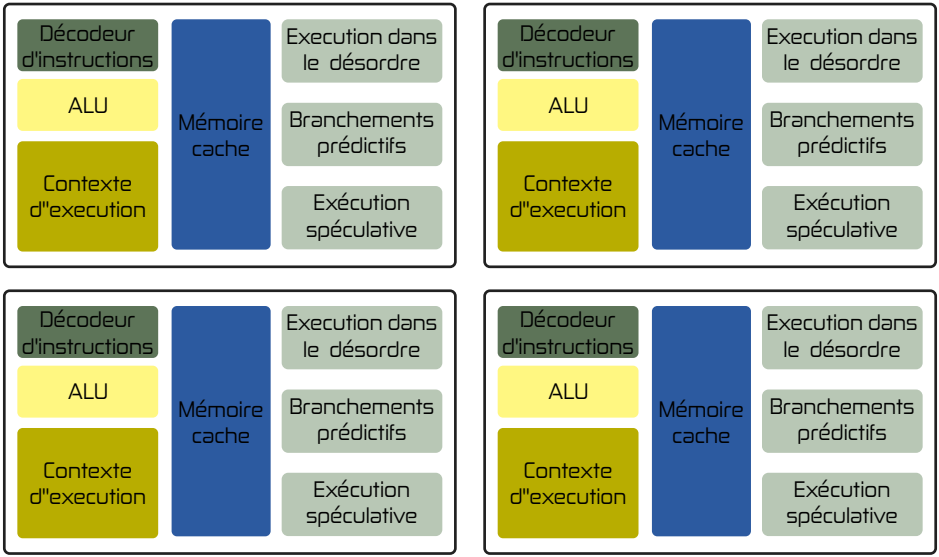
Les GPU utilisent un modèle d'execution SIMT

- SIMT = Single Instruction Multiple THREADS
- Sur un principe proche du SIMD, chaque «groupe» de threads execute la même instruction.
- On retrouve la principale limitation du SIMD : la divergence au sein d'un «groupe» de threads est très couteuse.
- Contrairement au SIMD plusieurs mécanismes permettent le partage d'informations au sein d'un «groupe» de threads et au delà.

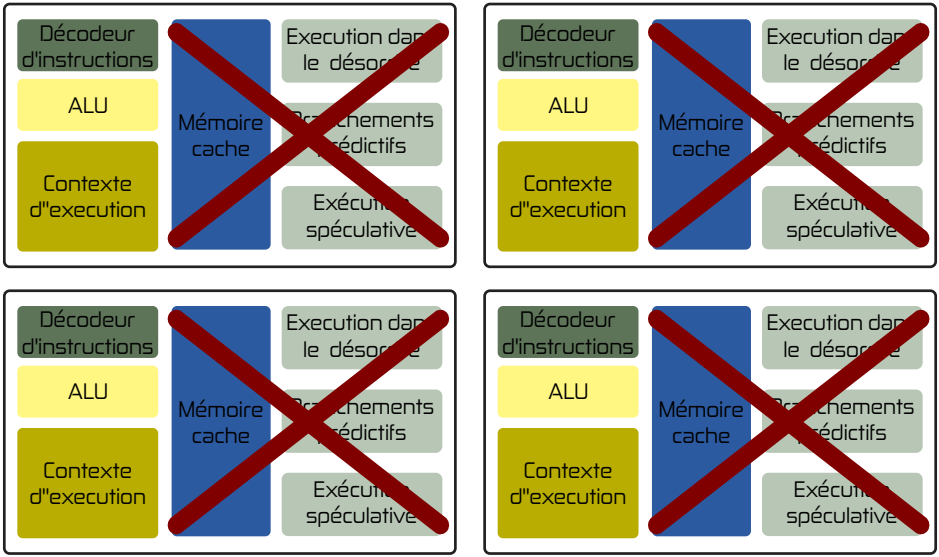
Un CPU est un tout terrain «cher»



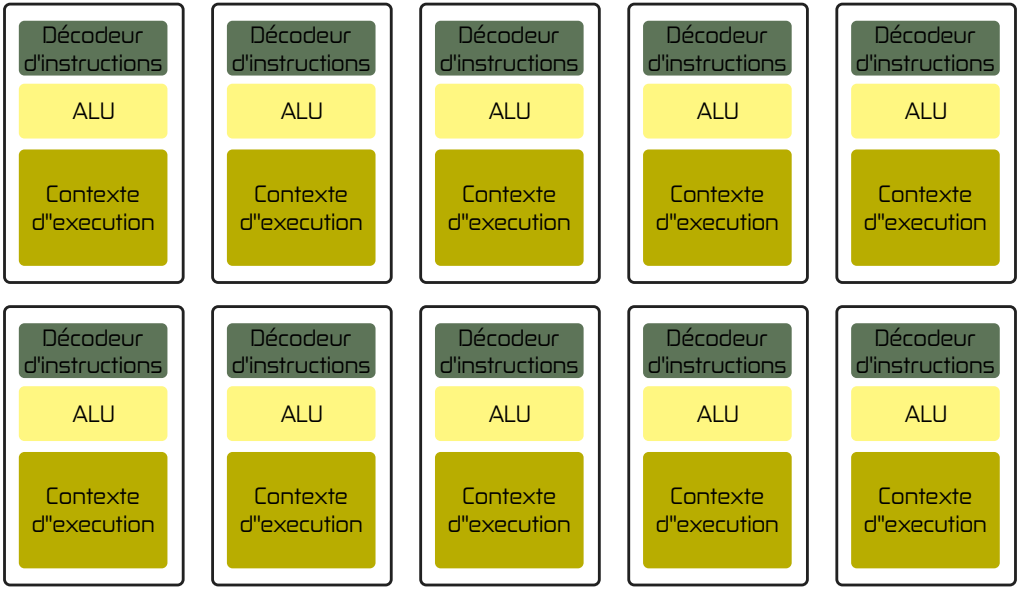
Un CPU multi-cœur est une équipe de tout terrains «chers»



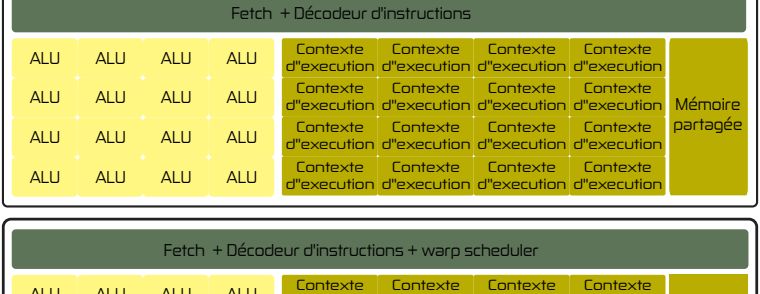
Les GPU contiennent des «CPU» très basiques

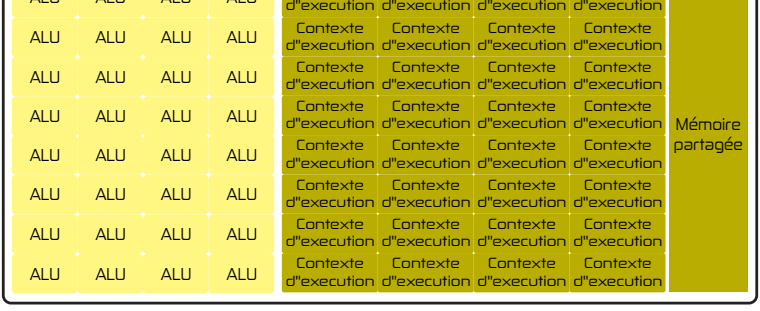


Mais peuvent en contenir beaucoup plus,

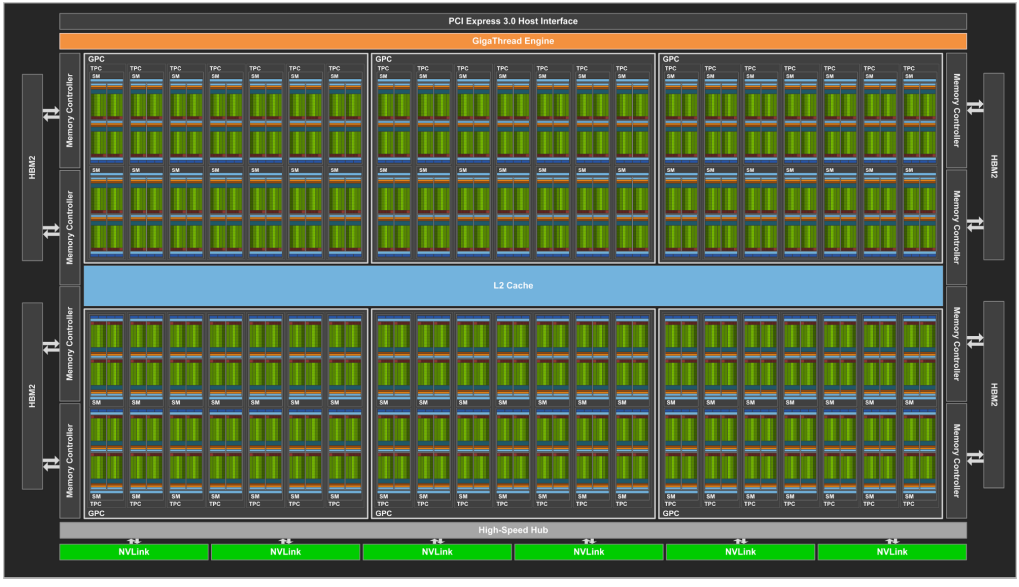


partager les coûts de récupération et de décodage d'instructions,

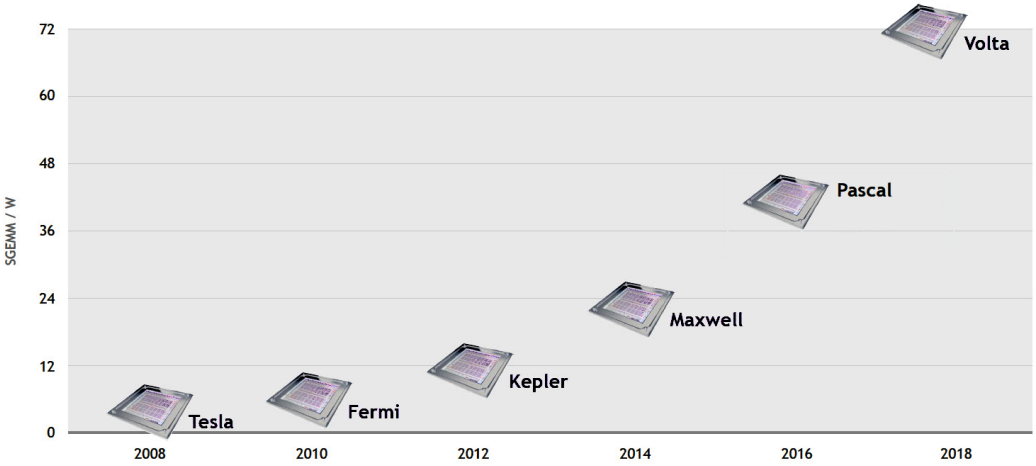




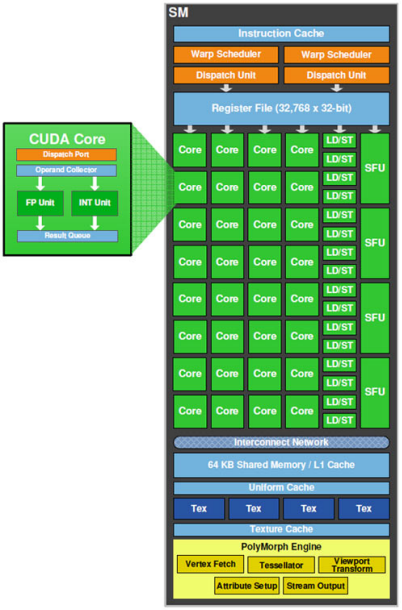
et en contenir vraiment plus !



Evolutions Fermi, Kepler, Maxwell, Pascal, Volta, Turing, ...



L'unité de base réelle est le Streaming Multiprocessor (SM)



Streaming Multiprocessor (SM)

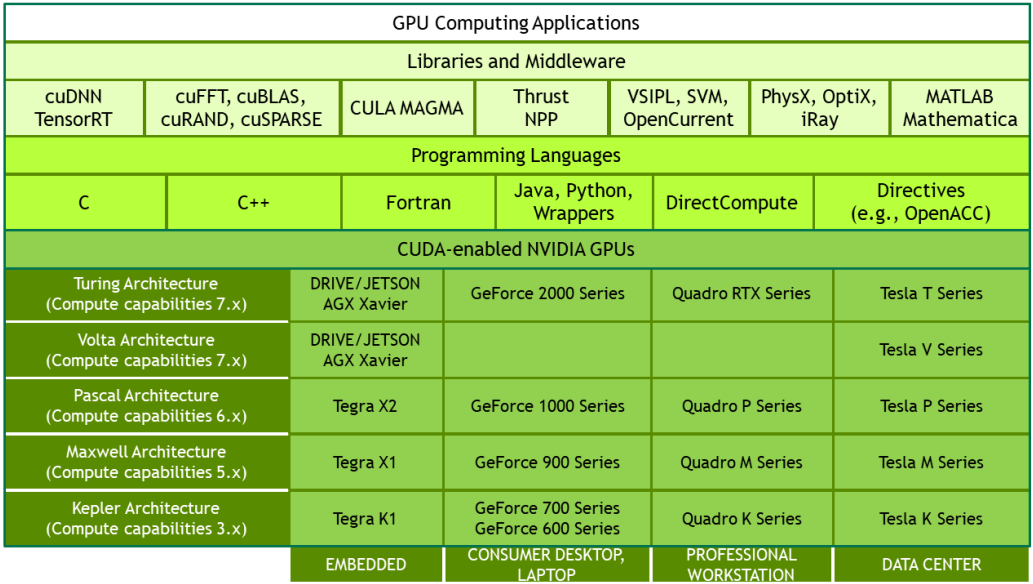
- Pour la génération Fermi Un SM contient 32 cœurs CUDA et quatre unités de textures
- Un premier niveau de cache est partagé entre chaque unité du SM
- Le travail est distribué par les deux «Warp scheduler». Chaque scheduler peut envoyer une instruction à 16 processeurs CUDA à chaque cycle d'horloge.

À chaque génération le SM est remanié (exploration architecturale)

Modele de programmation des GPU : une histoire de CUDA

- Durant sa thèse, Ian Buck avec son équipe à Stanford développe en 2004 «Brook for GPUs», une API qui permet d'utiliser le GPU pour «streamer» des calculs vers le GPU. Ils inventent alors le GPGPU (General Purpose GPU).
- Depuis Ian Buck est «general manager and vice president of Accelerated Computing at NVIDIA» et Brook for GPUs est devenu CUDA (Compute Unified Device Architecture).

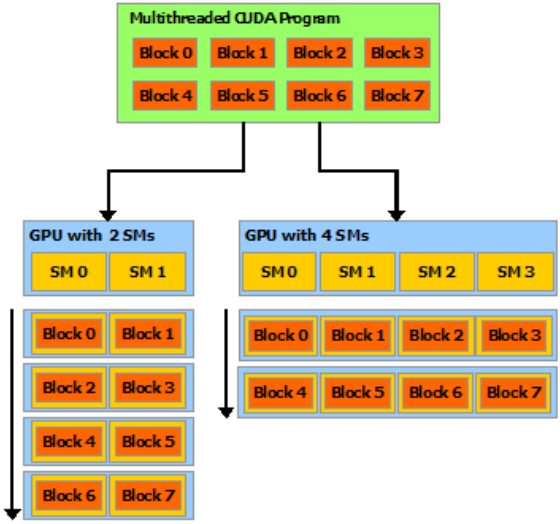
CUDA : bien du chemin parcouru



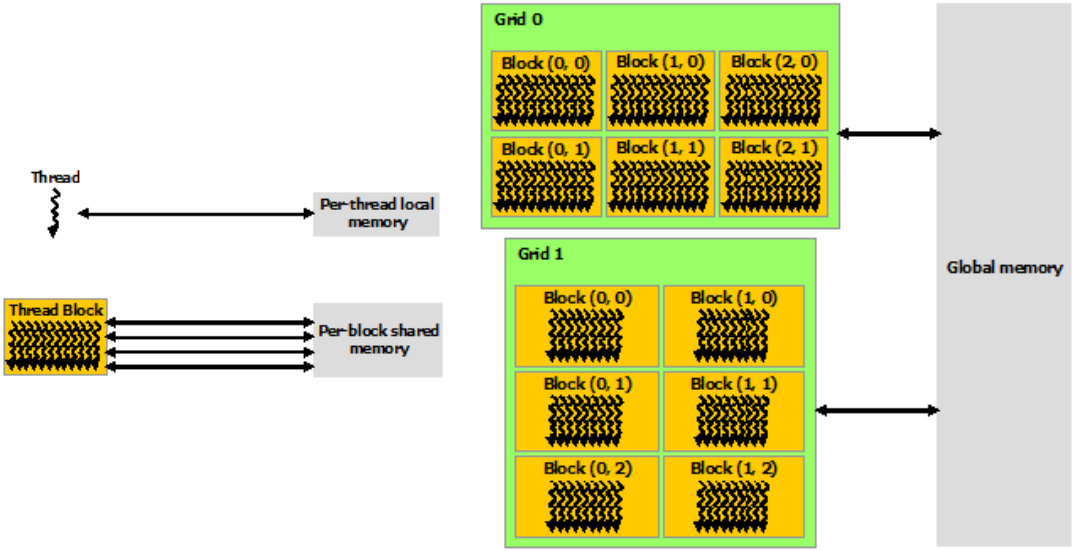
CUDA : un modèle de programmation basé sur des abstractions

CUDA exploite un modèle de programmation basé sur des abstractions plutôt que sur l'architecture réelle. Cela permet de faire passer à l'échelle le programme.

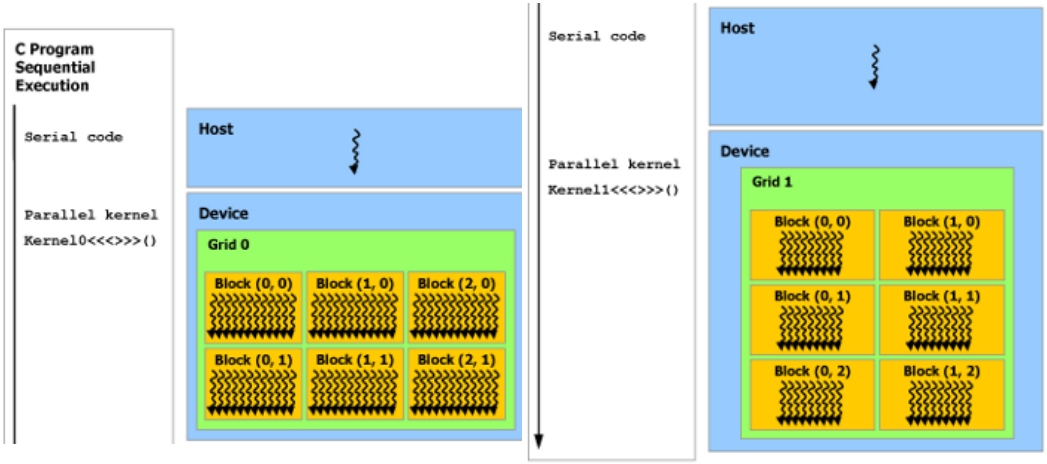
- la hierarchie des threads
- les mémoires partagées
- les barrières de synchronisation



CUDA : hierarchie des threads et des mémoires

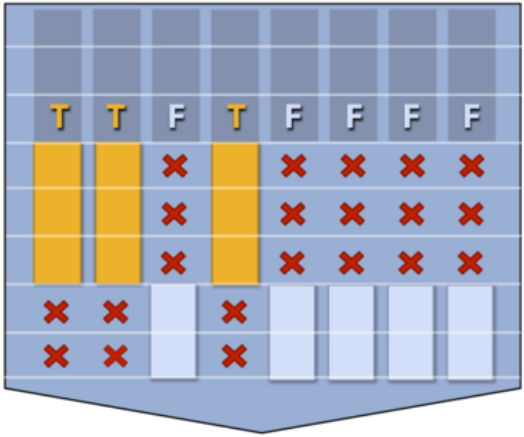


CUDA : hierarchie des threads et des mémoires



CUDA : Structures conditionelles

À défaut de pouvoir laisser les threads diverger, si lors d'un test l'ensemble des thread ne prends pas la même branche alors les deux branches sont executées. Les performances sont alors mauvaises.



CUDA : Exemple de Kernel (Device code)

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

int main()
{
// pas mal de lignes avant
// Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
// pas mal de lignes après
}
```

CUDA : Appel d'un kernel

Pour appeler le kernel (fonction executée sur le GPU) avec n_1 blocs de n_2 threads. il suffit d'invoquer :
NomDuKernel«<n1, n2»>(paramètres de la fonction)
Exemples :

- `<<<1, 256>>>` : 1 bloc de 256 threads
- `<<<256, 1>>>` : 256 blocs de 1 thread chacun
- `<<<16, 16>>>` : 16 blocs de 16 threads chacun

Il faut bien sur $n_1 \times n_2 \geq N$ avec N la taille du problème à résoudre.

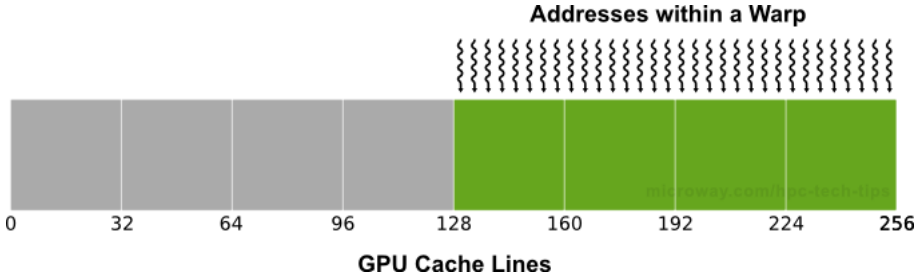
CUDA : Variables prédéfinies

- uint3 threadIdx : coordonnées du thread dans le bloc
- uint3 blockIdx : coordonnées du bloc dans la grille
- dim3 blockDim : dimension du bloc
- dim3 gridDim : dimension de la grille
- int warpSize : nombre de threads dans le warp

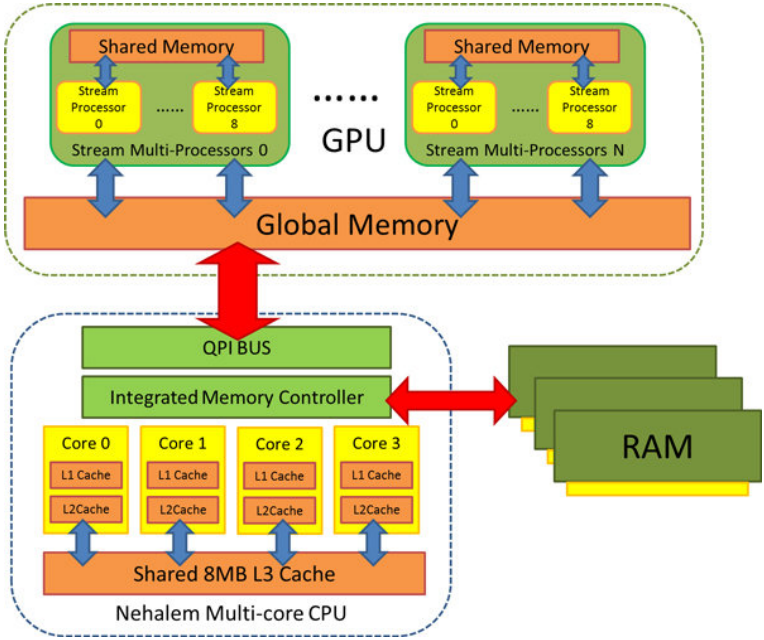
```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

CUDA : Coalescence

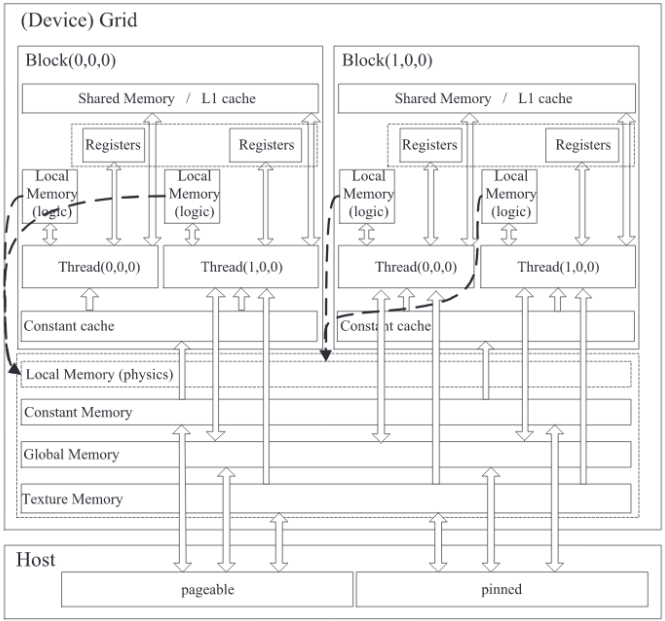
La mémoire globale est lente comparativement aux performances du GPU. Afin de l'utiliser à son maximum, tous les threads d'un même WARP doivent charger des adresses consécutives. Les accès sont alors dits coalescents.



Communications CPU GPU



Hierarchie mémoire



Atomics

https://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf