Nvidia CUDA

Organisation générale

- 1 CM (05/09) – 1h15
- 2 TPs (05-06/09) – 4h
- Examen : QCM (23/09)

Agenda

1. Introduction et Enjeux
2. Architecture
3. Modèle de programmation
4. Modèle d’exécution
5. C/C++ API

Agenda

1. Introduction et Enjeux
2. Architecture
3. Modèle de programmation
4. Modèle d’exécution
5. C/C++ API

1 – NVIDIA CUDA

Compute Unified Device Architecture

1. Cartes graphiques Nvidia
2. + pilotes CUDA
3. + extensions de langage C/C++
4. + compilateurs, outils, libraries.

1 – Environnement Logiciels

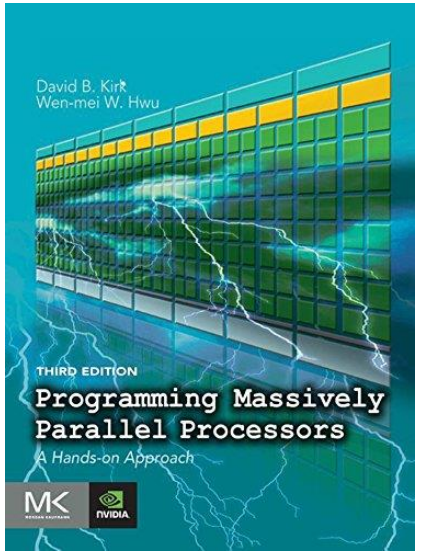
Téléchargement sur le site

https://developer.nvidia.com/cuda-downloads

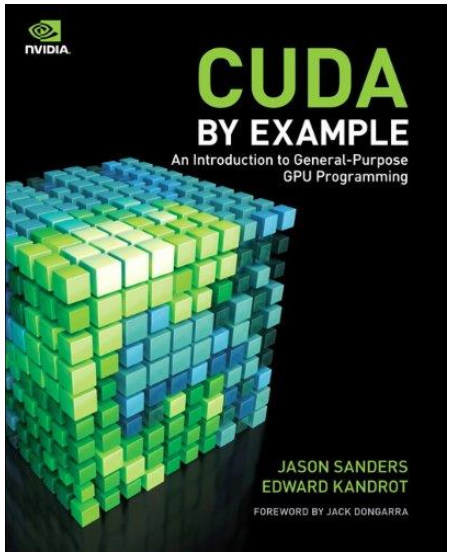
1. Pilotes Nvidia
2. CUDA Toolkit : compilateur, debugger, documentation, exemples
3. Nvidia Nsight (Visual Studio ou Eclipe)
4. Bibliothèques : CuBLAS, CuFFT …

1 - Ressources

- David B.Kirk et Wen-mei W. Hwu : *Programming Massively Parallel Processors*



- Jason Sanders et Esward Kandrot : *CUDA by Example*



1 - Ressources

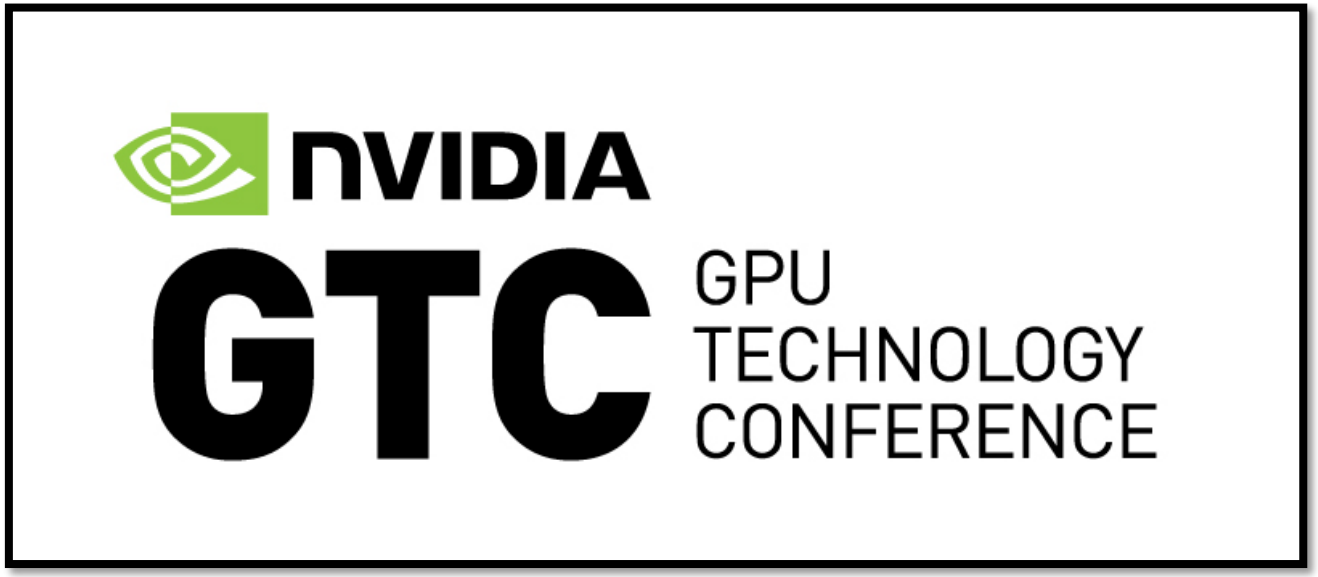
- Nvidia Developer et CUDA zone

✓ *https://developer.nvidia.com/cuda-zone*

- Conférence annuelle GTC

✓ *https://www.nvidia.com/fr-fr/gtc/*

✓ *19-22 Sept*



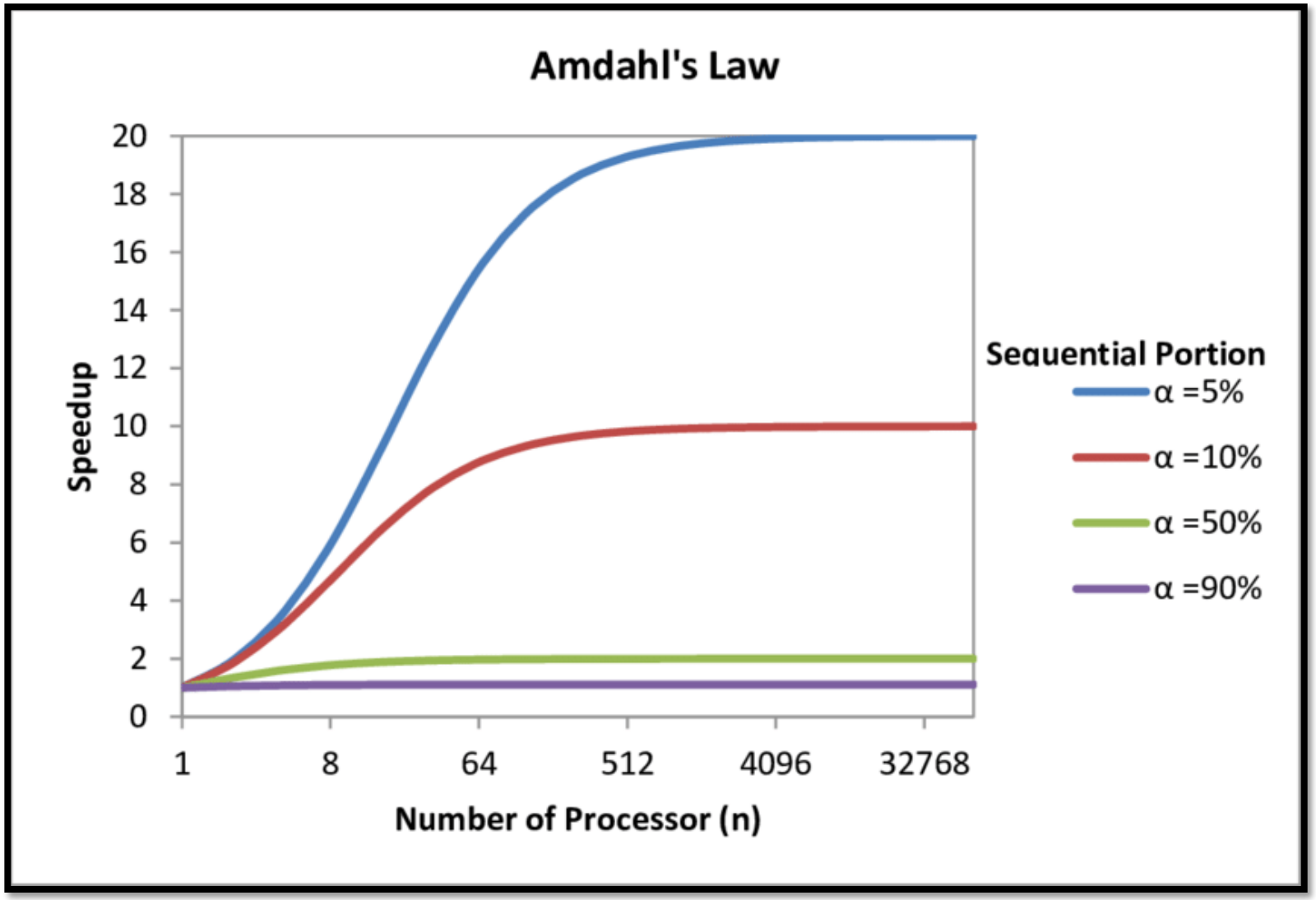
1 – Domaines applicatifs

- 3D, IA, Simulations, Metaverse ….

1 – Taxonomie de Flynn

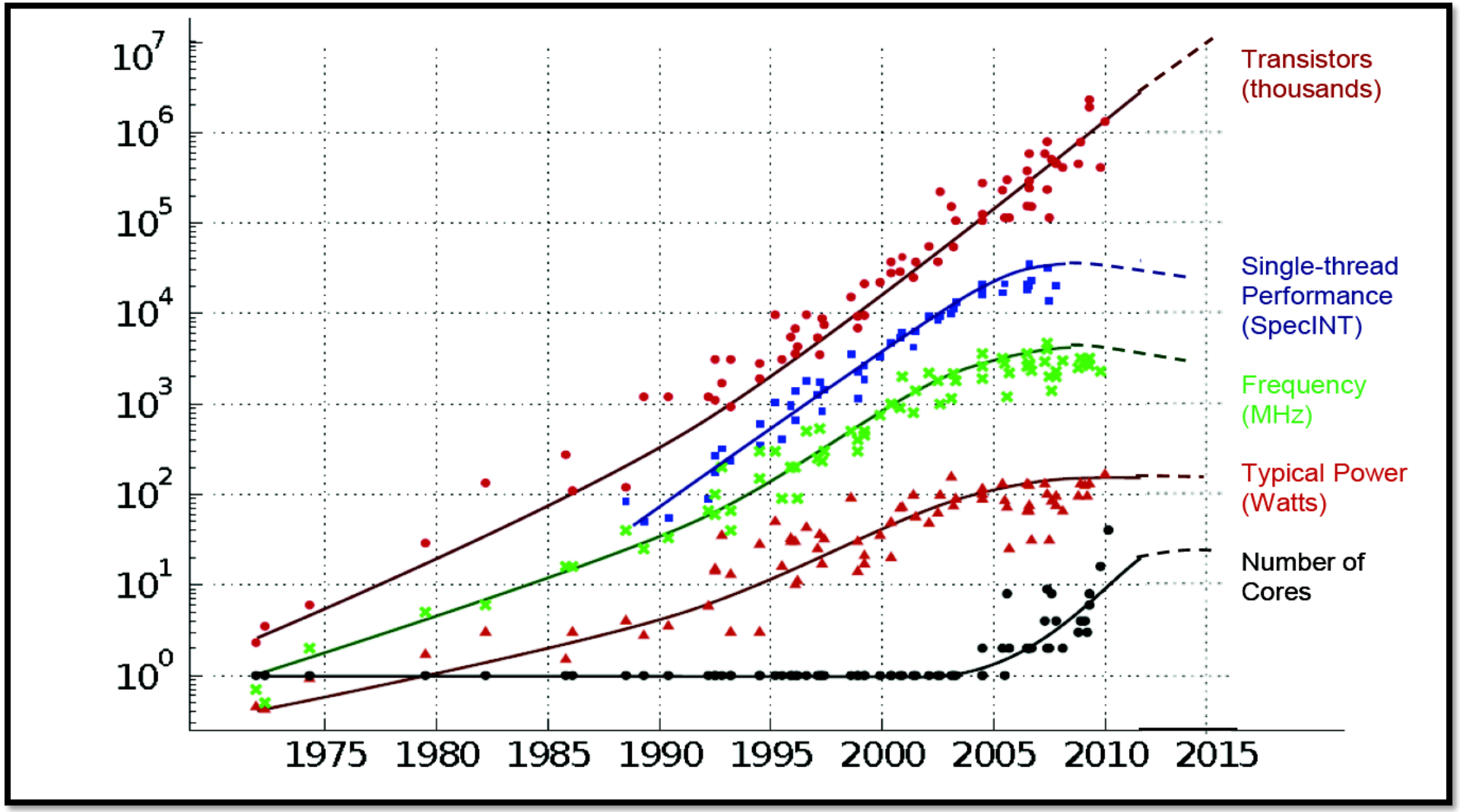
	Single Data	Multiple Data
Single Instruction	**SISD**	**SIMD**
Multiple Instruction	**MISD**	**MIMD**

1 – Loi Amdhal

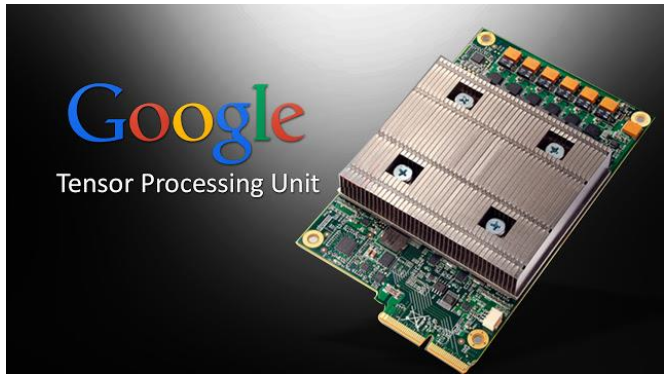


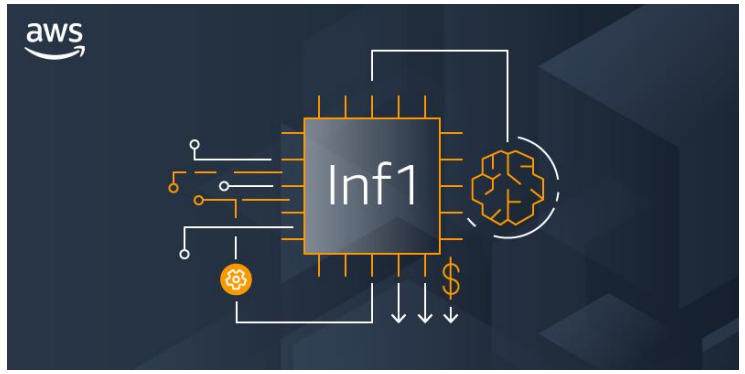
$$Speedup = \frac{1}{(1-p) + p/N}$$

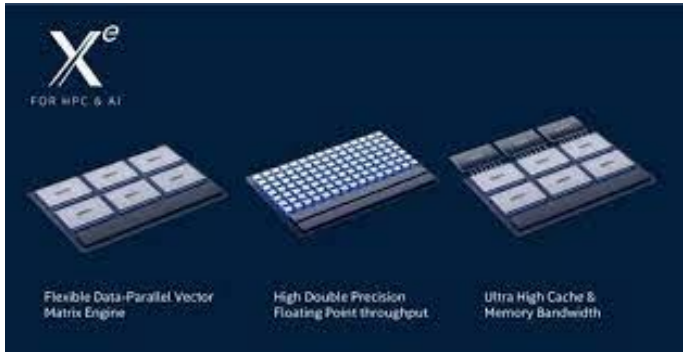
1 – Loi de Moore

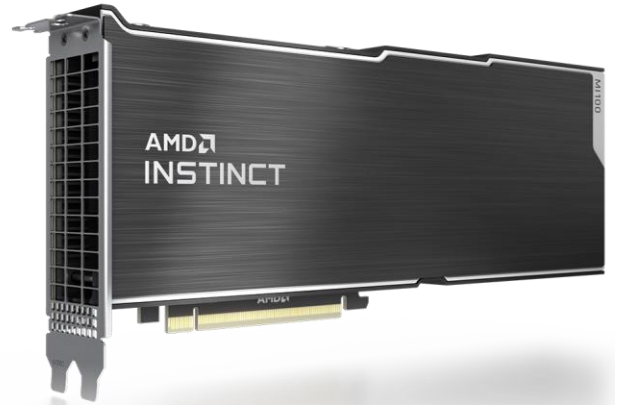


1 – Pourquoi les accélérateurs ?

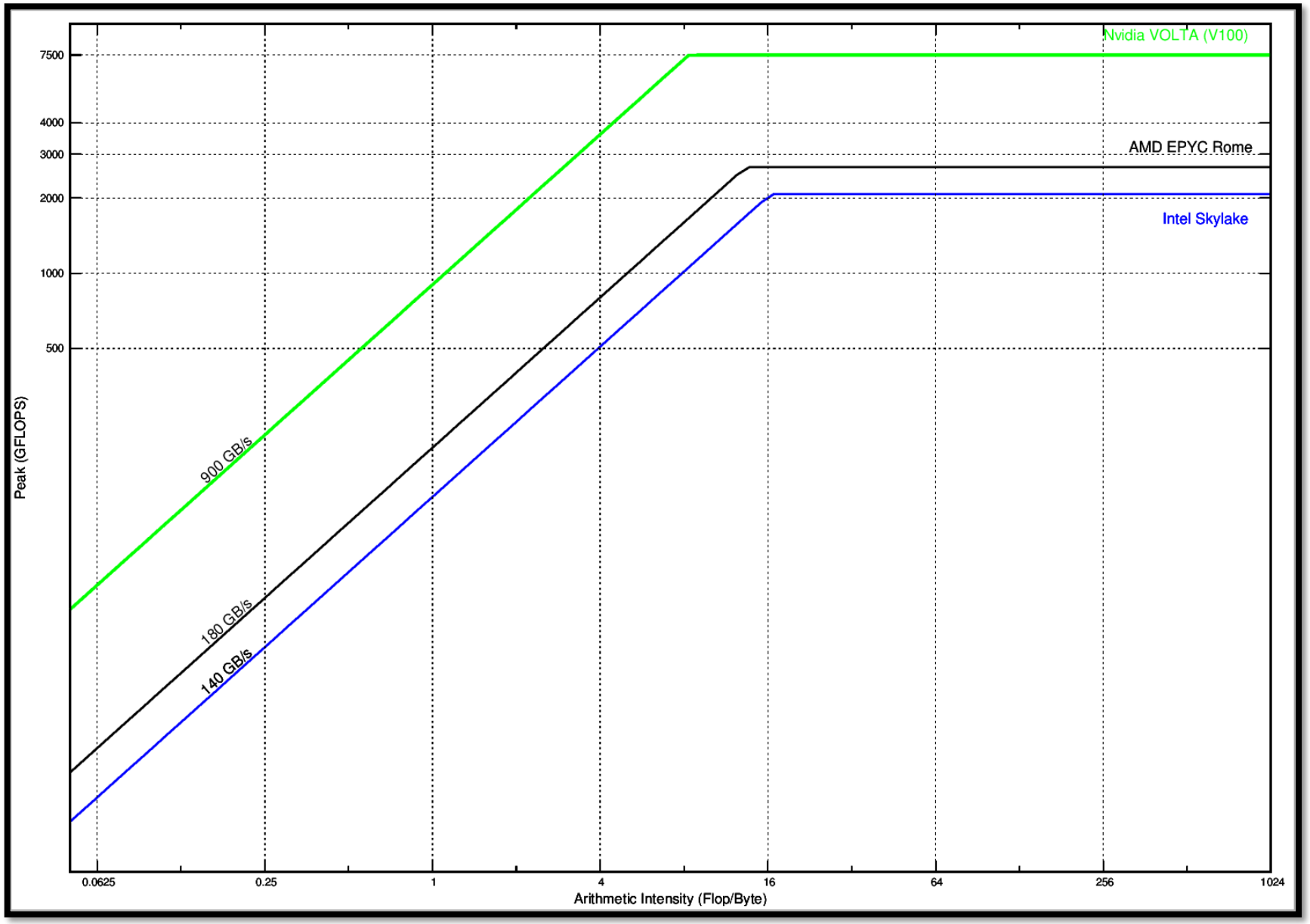








1 – Pourquoi les GPUs ?



1 – Gammes Nvidia

- GeForce : gamme grand public (jeu)
- Quadro : gamme professionnelle pour la 3D
- Tesla : gamme professionnelle pour GPGPU : pas de composants video, mémoire ECC.
- Tegra : gamme pour les plateformes nomade (tablettes, smartphones, voitures autonomes)

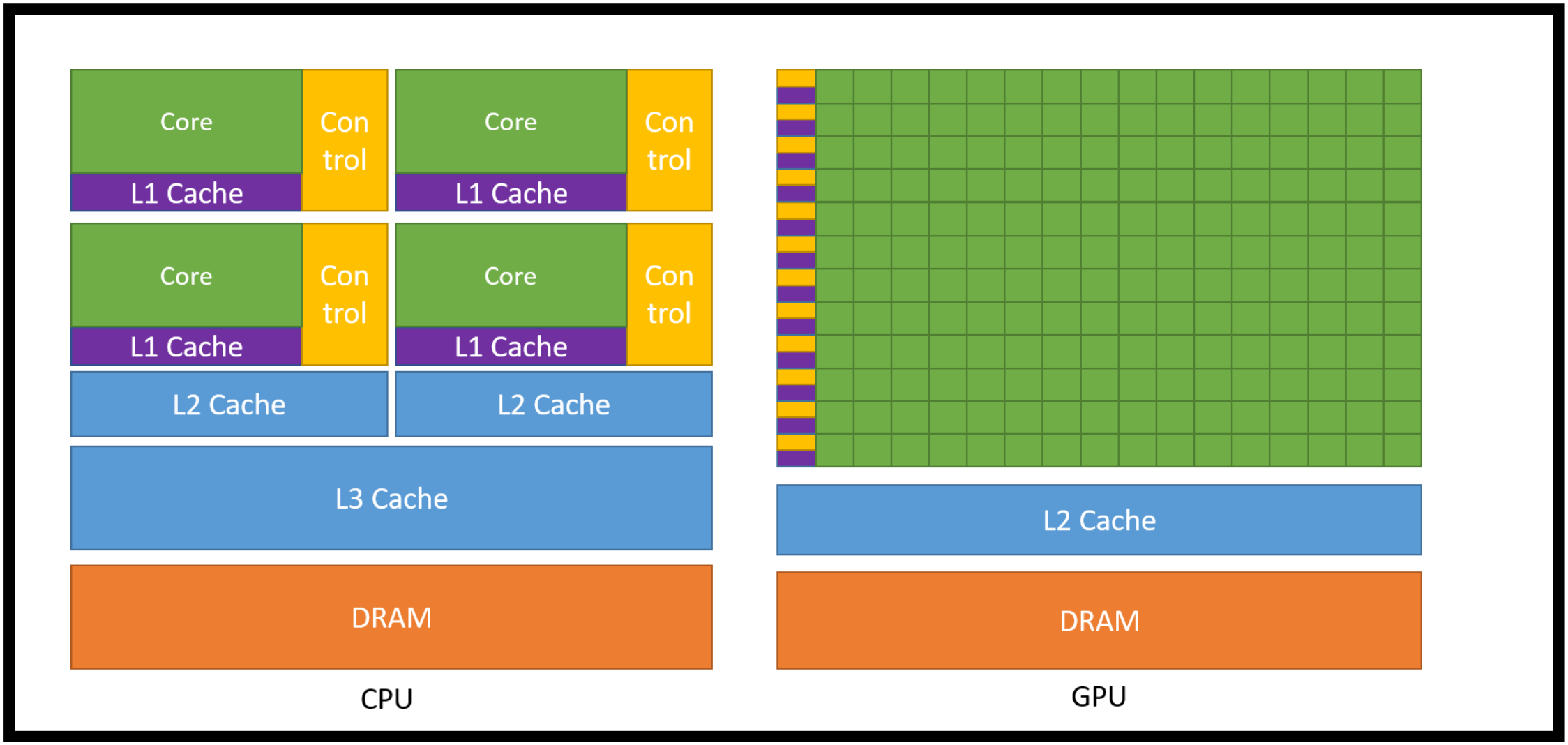
1 – Bilan

- Nvidia CUDA : plus large qu'une API pour les GPUs
- Environnement complet : outils, librairies …
- Programmation simple, optimisation itérative
- Diversité des accélérateurs

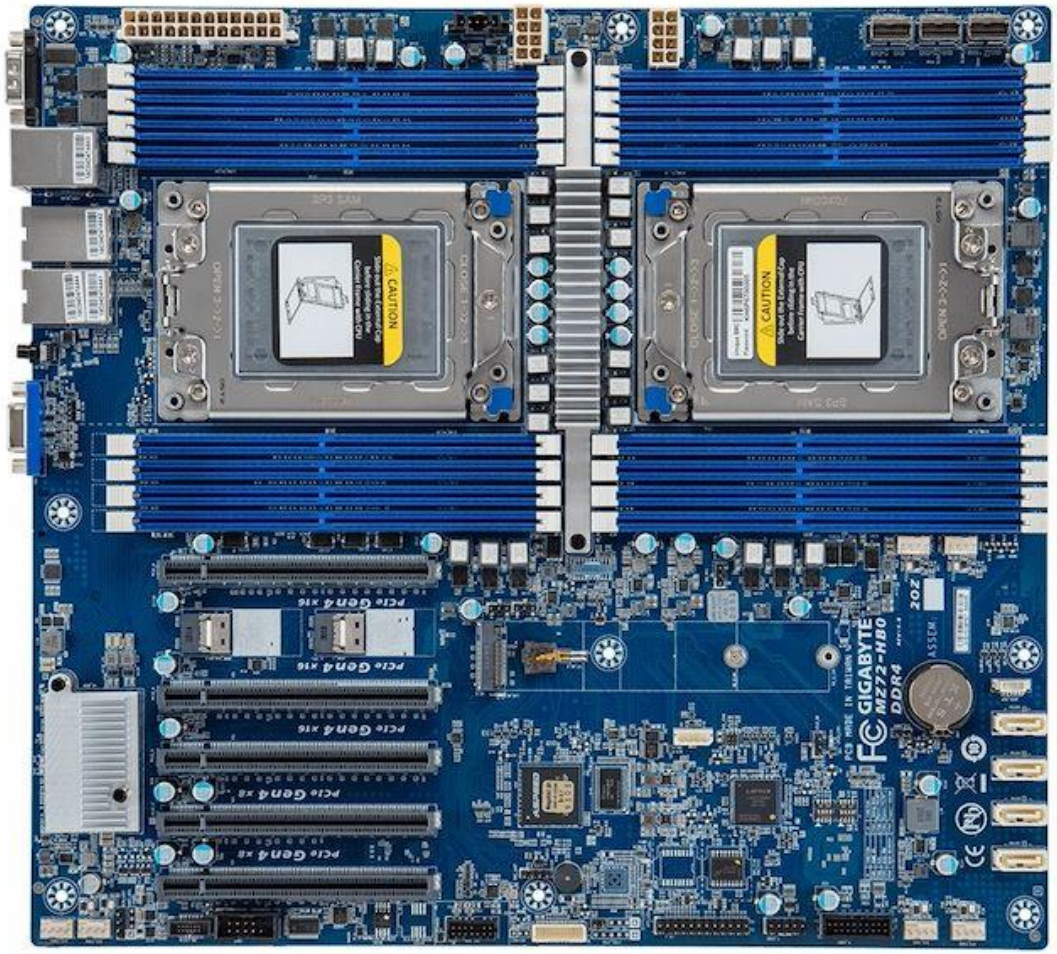
Agenda

1. Introduction et Enjeux
2. Architecture
3. Modèle de programmation
4. Modèle d’exécution
5. C/C++ API

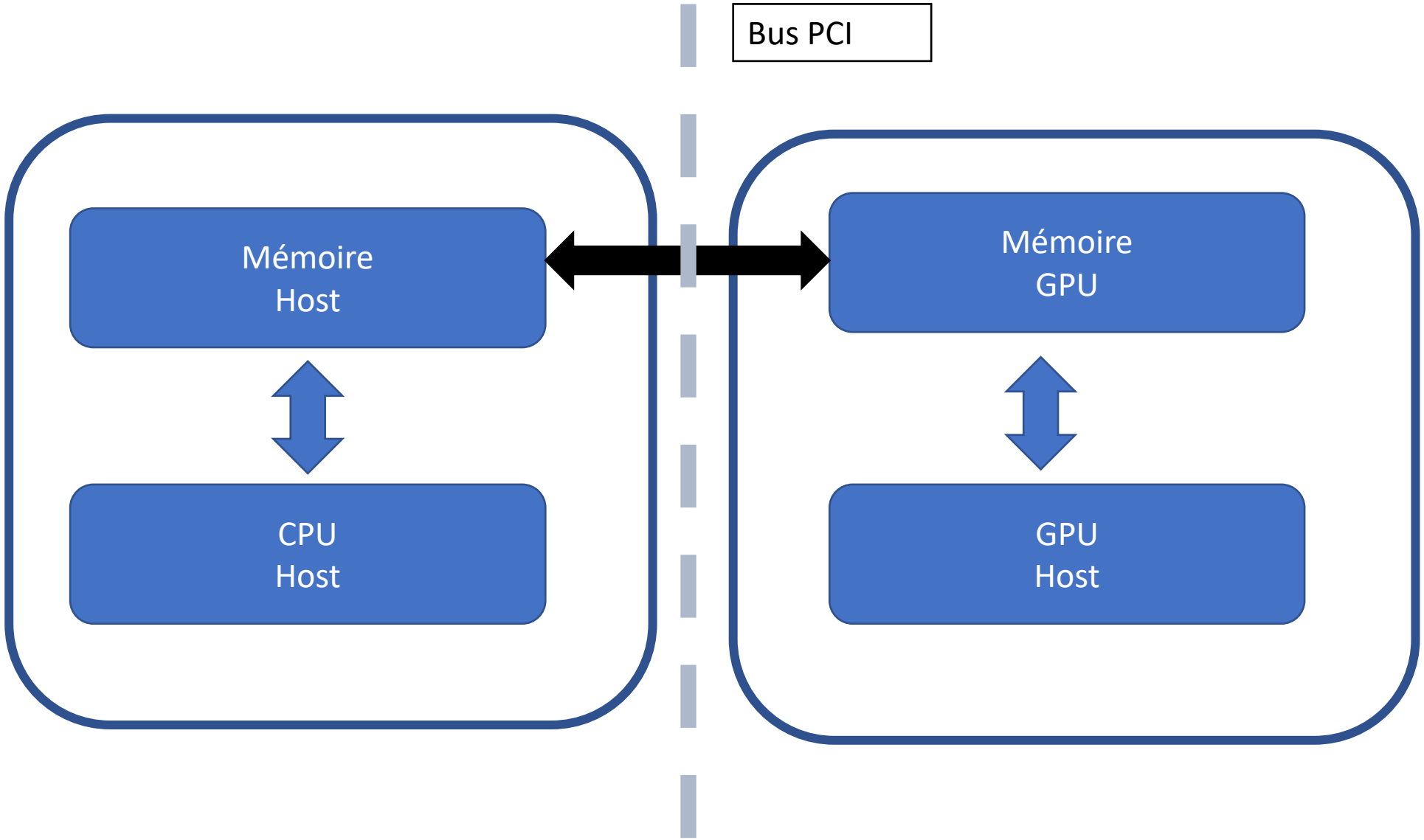
2 – Architecture générale



2 – Architecture générale



2 – Architecture générale



2 – Générations de GPU

Générations	Année
Tesla *Cuda*	*2008*
Fermi	*2010*
Kepler *GPU Boost, GPUDirect, Dynamic Parallelism*	*2012*
Maxwell	*2014*
Pascal *HBM2, Nvlink*	*2016*
Volta *Tensor Cores*	*2017*
Turing	*2018*
Ampere	*2020*

Compute Capabilities

https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

2 – Mémoires

Dans le GPU : mémoires rapides mais de tailles limitées

- Registres
- Mémoire partagée, gérée par le développeur
- Caches : mémoire de type constante et texture

Dans la DRAM : mémoire plus lente mais en grande quantité

- Mémoire locale et globale
- Mémoire de type constante et texture

2 – Mémoires

Type	On-chip	Cached	Accès	performance
Registres Shared	oui oui	- -	rw rw	1 cycle 1 cycle
Local Global Constant Texture	non non non non	non non oui oui	rw rw r r	lent (~100 cycles) lent (~100 cycles) 1 .. 10 .. 100 1 .. 10 .. 100

2 – Unités de calcul

- Cuda core - Streaming Processor
  - ✓ Unité Arithmétique
  - ✓ Registres

- Cuda Streamings Multiprocessors (SMs)
  - ✓ Attaché à la mémoire globale
  - ✓ Plusieurs Cuda Core

- Cuda GPU
  - ✓ Plusieurs Streaming Multiprocessors (SMs)
  - ✓ Mémoire globale

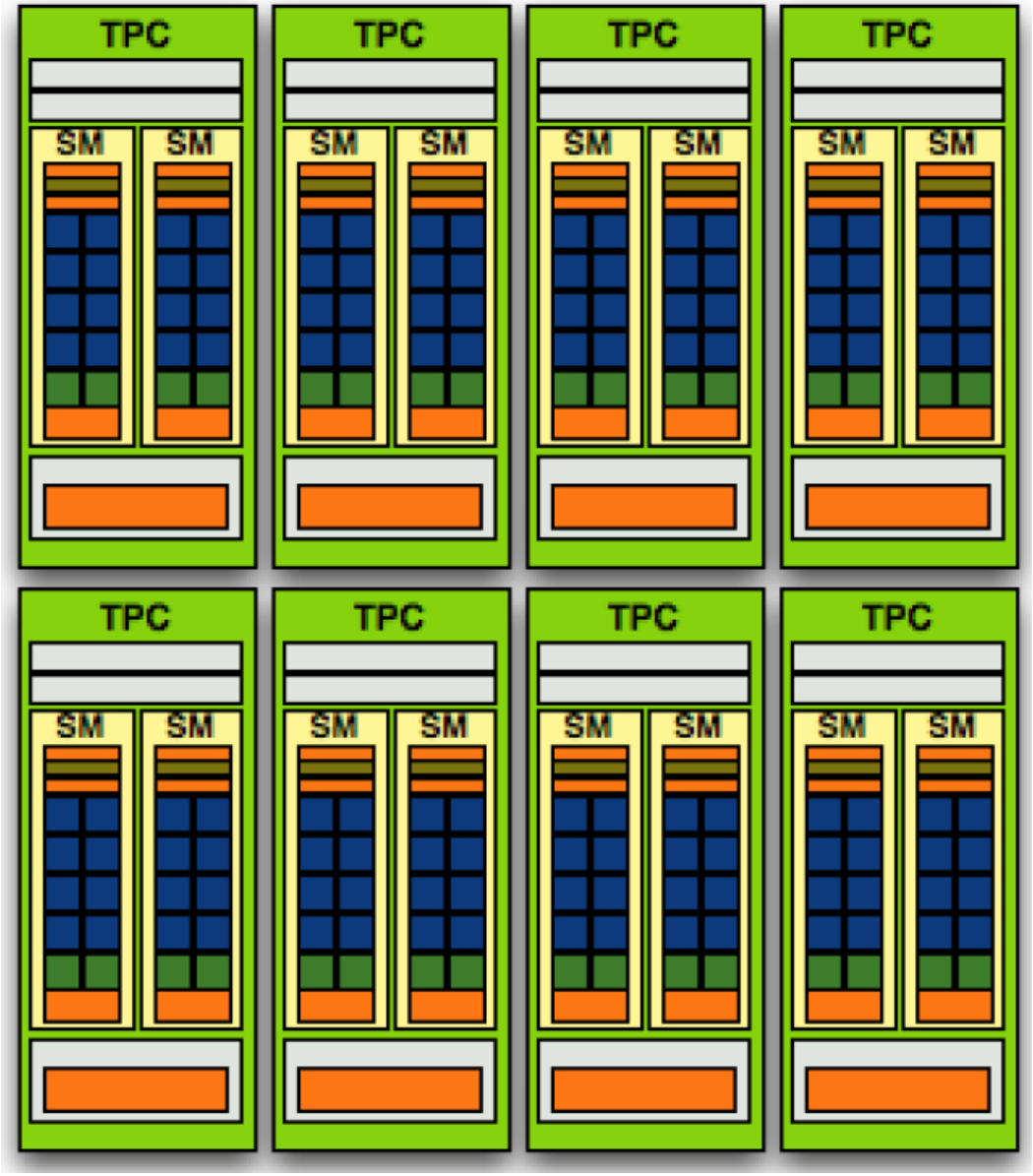
2 – Architecture Tesla (G80)

- Unités de calcul regroupées par 8

→Multi-Processeurs (SMs)

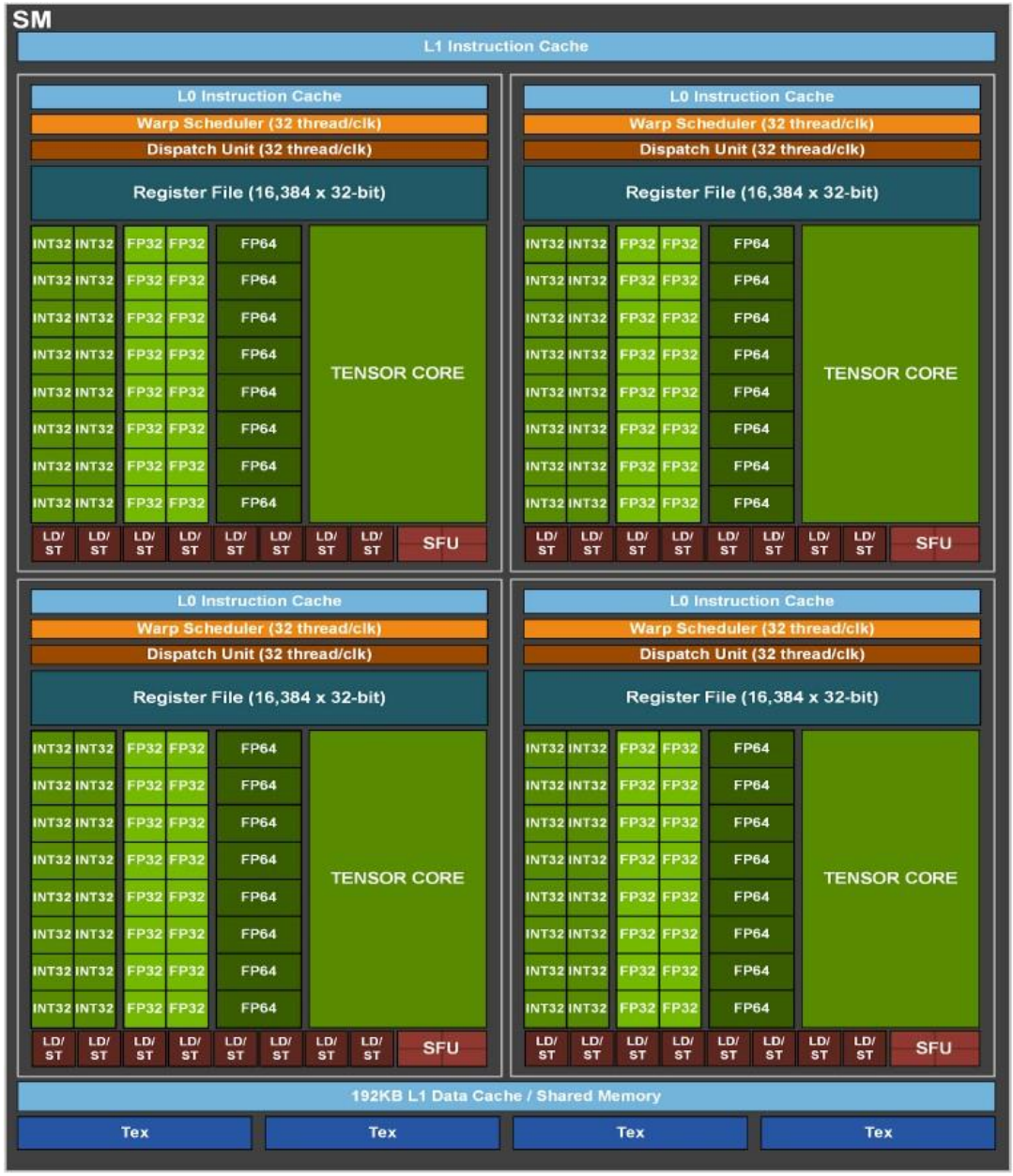
- Multi-Processeurs regroupés par 2 (G80)

→ Texture Processing Clusters (TPC)

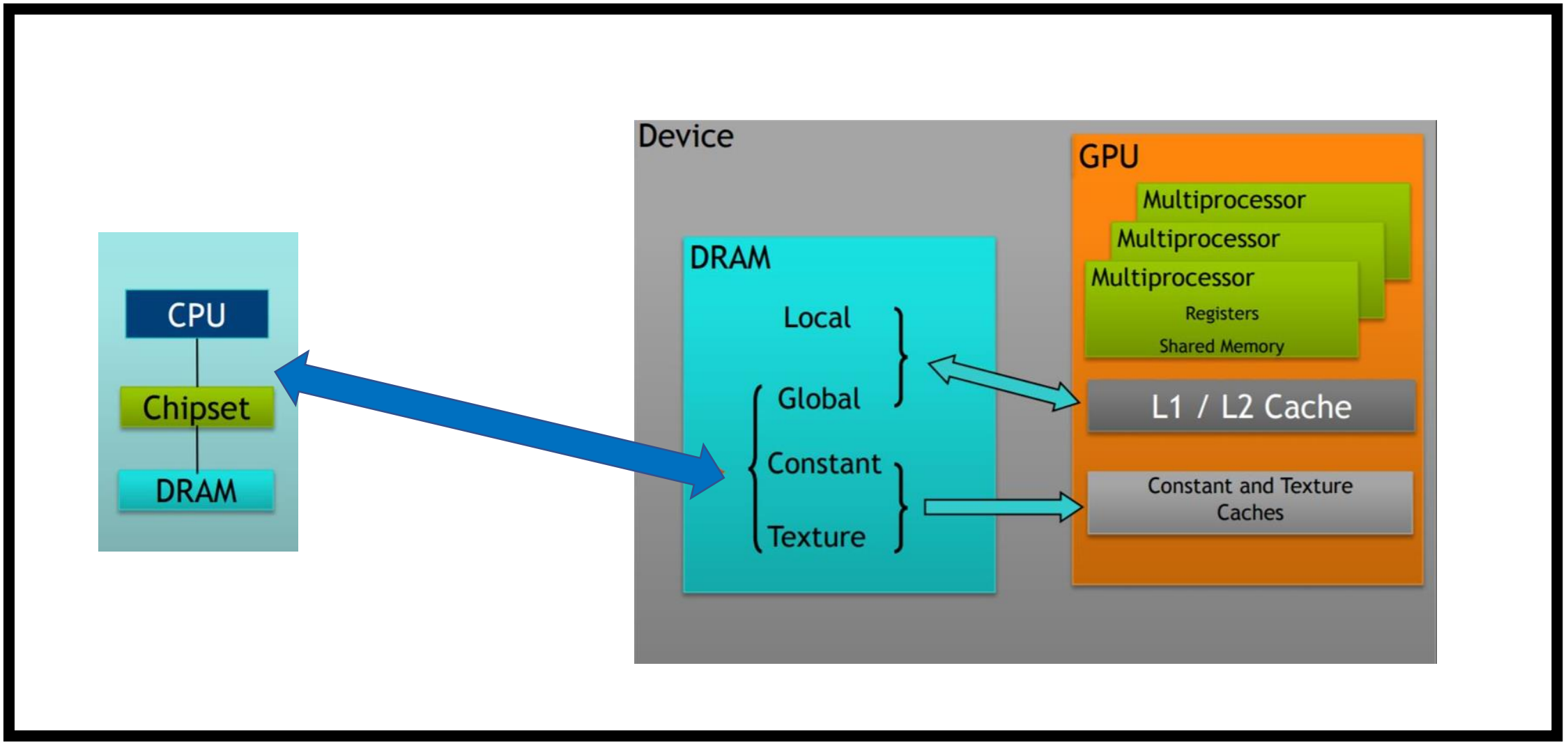


2 – Architecture Volta (V100)

- Unités de calcul entier, simple et double precision
- Tensor cores
- 80 SMs , +5000 CUDA cores
- Mémoires : HBM2, L2 cache



2 – Mémoires / Unités de Calcul



2 – Architectures : comparatif

Modele Tesla	K80 Kepler	M60 Maxwell	P100 Pascal	V100 Volta	A100 Ampere
Bus	PCI	PCI	PCI	PCI	PCI
Fréquence (MHz)	560	899	1126	1372	1265
Streaming Multiprocessors	2x13	2x16	56	80	108
FP32 cores	4992	4096	3584	5120	6912
FP64 cores	-	-	1792	2560	3456
Tensor cores	-	-	320	432	432
Global Memory	24	16	16	32	40
FP32 (Teraflops)	8.74	9.6	10.6	14.0	19.5
TF32 (Teraflops)	-	-		125	312
Bande Passante (GB/s)	480	320	720	900	1555
Power (Watts)	300	250	250	250	400

2 – Bilan

- Plusieurs générations : Tesla, Fermi, Kepler …
- Optimisations spécifiques à chaque architecture.
- Connaissance approfondie des architectures et de leur fonctionnement.

Agenda

1. Introduction et Enjeux
2. Architecture
3. Modèle de programmation
4. Modèle d’exécution
5. C/C++ API

3 – Programmation hétérogène

Un programme contient à la fois :

- Un code **host**, qui sera exécuté par le CPU,
- Le code **device** ou **kernel**, qui sera exécuté par le GPU.

Code host

- l'exécution du code **device**
- les communications entre la mémoire **host** et **device.**

Code device ou Kernel

Un **kernel** est une procédure (pas de valeur retour) → exécution par les cœurs du GPU (portion parallèle de programme).
Un kernel peut appeler un autre kernel

3 – Coté device : Modèle SPMD

Parallélisme de données

- Modèle SPMD : Single Program on Multiple Data
- A l'exécution, le code host va instancier un kernel

Thread

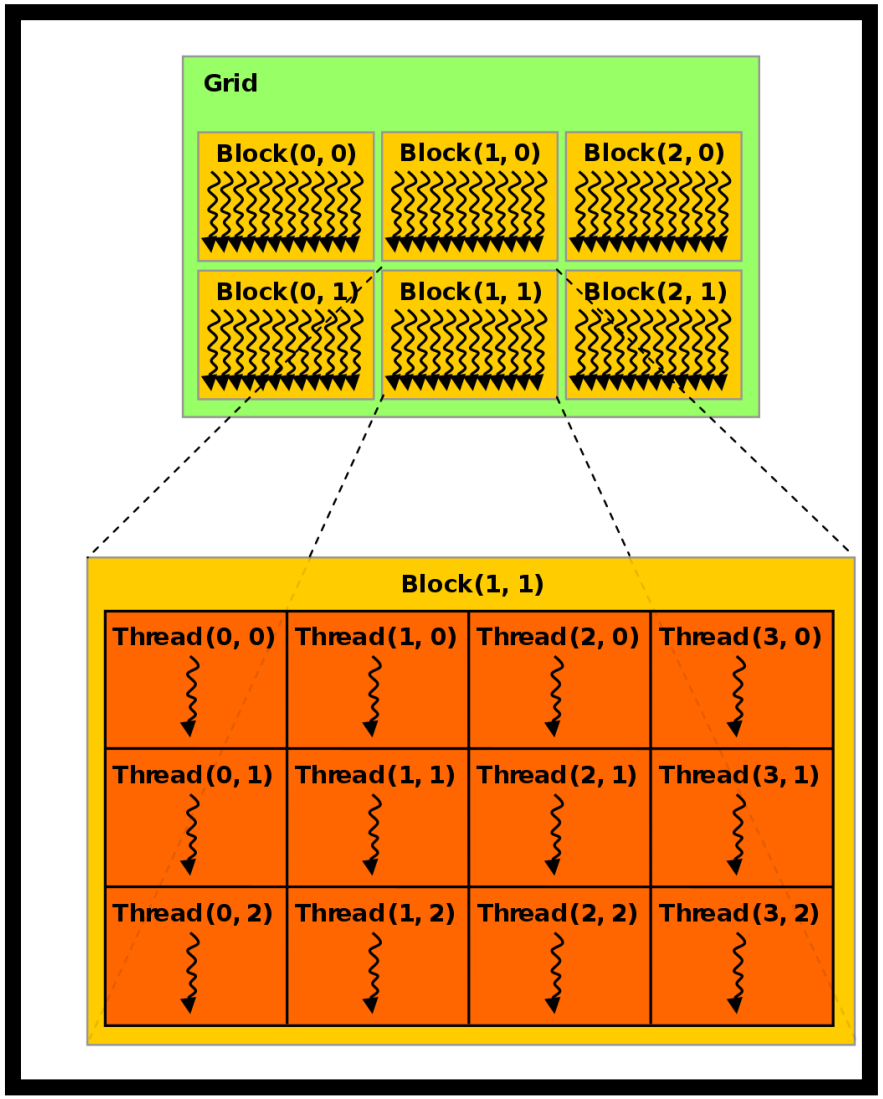
- Une instance de kernel est appelé thread *(différent d'un thread POSIX)*
- Grand nombre (plusieurs 100s/1000s)
- Tous les threads d'un même kernel exécutent le même code mais peuvent prendre des chemins différents en cas de blocs conditionnels.
- Tous les threads ne s'exécutent pas en même temps.

3 – Organisation des threads

- Les threads sont regroupés en blocs
- Les blocs sont groupés en une grille

⇒ *Exécuter un kernel : exécuter une grille de blocs de threads*

Chaque thread/bloc possède un identifiant (1D, 2D, 3D).



3 – Remarques

Choix du nombre de threads, de la topologie :

- Sur le CPU : nombre de threads correspondant u nombre de cœurs disponibles
- Sur le GPU : nombre de threads correspondant au nombre de données à traiter

Exemples:

- Somme de 2 vecteurs 1D de n flottants → n threads, topologie 1D
- Traitement d'une image de Nx.Ny pixels → blocs 2D *(tx, ty)* de threads
  On peut avoir Nx.Ny < tx.tyn (plus de threads que de pixels) traiter

3 – Déroulement d'un programme CUDA

1. Initialisation des mémoires
- Initialisation des données en mémoire sur le host
- allocation dans la mémoire global device : cudaMalloc

2. Copie des données de la mémoire host vers la mémoire : device cudaMemcpy

3. Exécution du kernel sur les données en mémoire device lancement des threads sur le GPU
- copie des résultats en mémoire device vers la mémoire host cudaMemcpy
- exploitation directe des résultats par OpenGL pour affichage

4. Libération de la mémoire globale device cudaFree

3 – Compilation

Le code host et device peuvent-être dans le même fichier
L'extension du fichier est .cu.
Le compilateur nvcc sépare les codes **host** et **device**.

- Le code **host** est compilé par le compilateur C/C++ par défaut
- Le code **device** est compilé par le compilateur CUDA puis le binaire est inséré dans l'exécutable

Agenda

1. Introduction et Enjeux
2. Architecture
3. Modèle de programmation
4. Modèle d’exécution
5. C/C++ API

4 – Code Host : Appel des fonctions CUDA

- Appels CUDA asynchrones, mais respect de la dépendance des appels
- **Recouvrement des communications** *(analogie MPI)*

- Temps d'exécution en utilisant **CUDAEvents** *(cf. TP)*

4 – Threads et processeurs

- L'exécution du kernel provoque la création de threads.

- Le placement des blocs de threads et des threads est effectué par le scheduler de la carte (politique de placement non documentée).

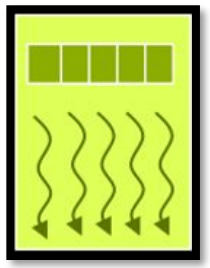
- Le **modèle d'exécution** définit la manière et les contraintes pour le placement.
  - *des blocs de threads sur les multiprocesseurs*
  - *des threads sur les cœurs*

- Chaque thread est exécuté par un processeur mais il faut tenir compte :
  - de l'organisation des threads en blocs
  - de l'organisation des processeurs en multi-processeurs

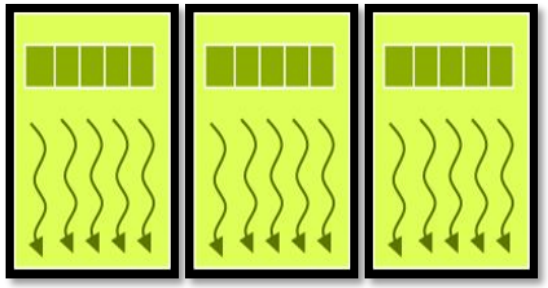
4 – Threads et processeurs - terminologie

Software

Thread



Thread block

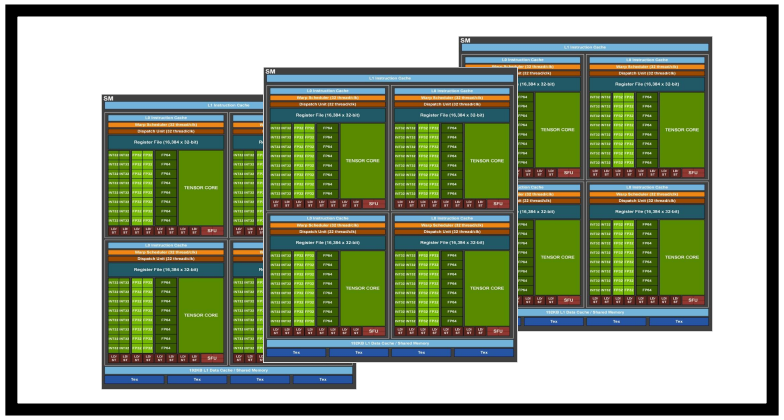


Grid

Hardware

Cuda Core

Multiprocessor



Device

4 – Règles

- Les threads d'un même bloc sont exécutés sur un même multiprocesseur.
- Un multiprocesseur peut se voir attribuer **plusieurs blocs**

- Le nombre de threads par bloc est limité (dépend de l'architecture – **e.g. 1024**).

- Les threads d'un même bloc sont exécutés :
  - instruction par instruction
  - par groupe de **32 threads** consécutifs → **un warp**

- ***Scheduler*** *:*
  - Niveau block et warp
  - Exposition au niveau de l'API

4 – Warp – exécution instruction

- Les threads d'un même warp sont exécutés **instruction par instruction**

- Une fois l'instruction exécutée sur les 32 threads, on recommence avec l'instruction suivante jusqu'à la fin du kernel.

⇒ **Optimisation**

- Pour optimiser l'utilisation d'un multiprocesseur il convient d'utiliser **des multiples** de 32 threads pour la taille des blocs, dans limite du nombre de threads par blocs.

- Exemple un GPU avec 16 multiprocessors et 32 cuda cores par multiprocessor
  - *16 blocs de 32 thread vs 8 blocs de 64 threads* ***(CUDA Occupancy)***

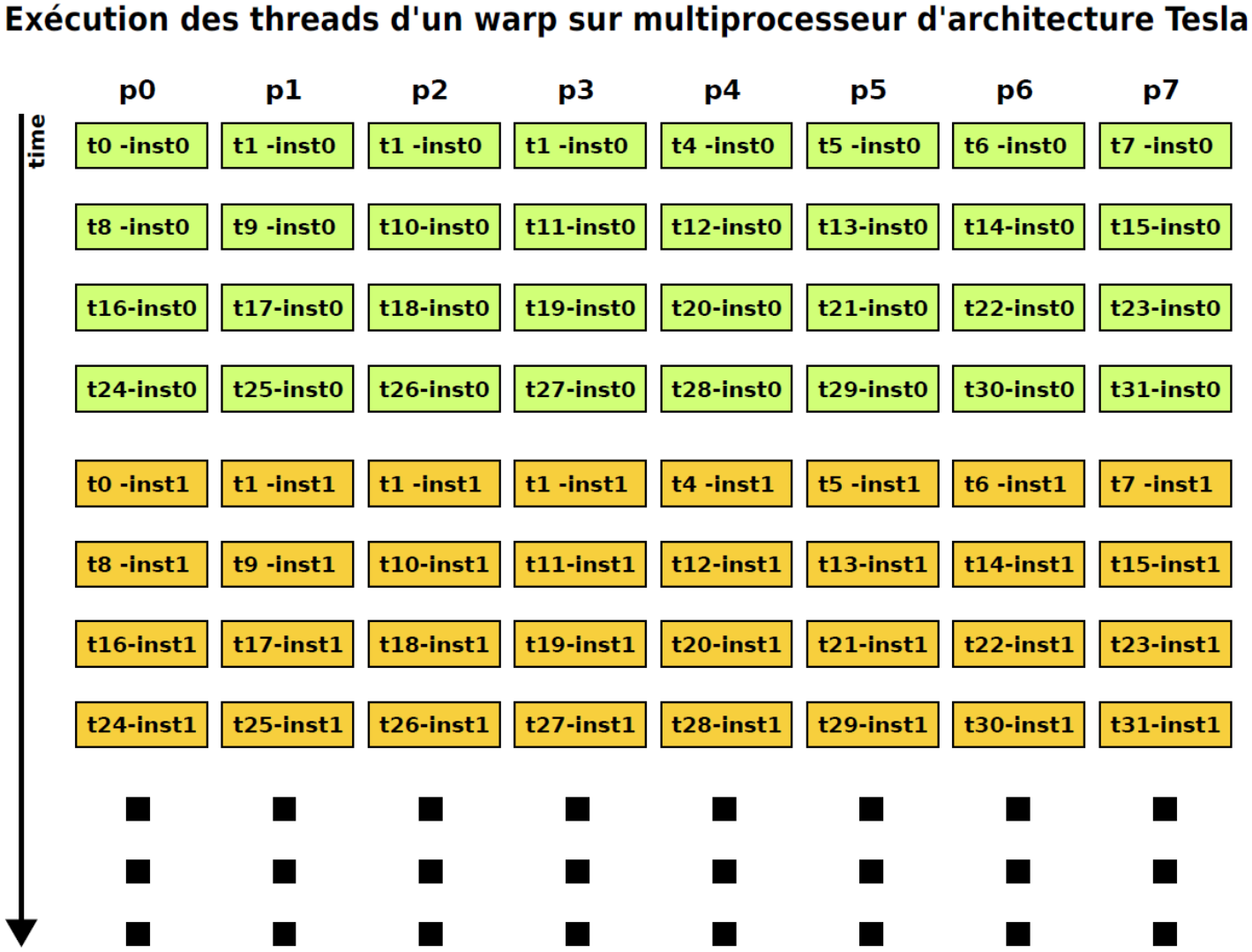
4 – Warp sur l'architecture Tesla

- Multiprocessor :

✓ *Diversité de ressources harware*

✓ *Instr fetch/decode/registres …*

- *Multiprocesseur (8 processeurs / cuda cores) exécute la première instruction du kernel sur les 8 premiers threads simultanément puis passe aux 8 suivants ….*



4 – Warp : Cas des structures conditionnelles

- Si des threads d'un même **warp** n'entrent pas dans la même branche de la structure conditionnelle, le modèle d'exécution **force l'évaluation séquentielle des deux branches**

- Les threads n'entrant pas dans une branche doivent attendre que les threads y entrant aient terminé leur exécution, puis inversement.
- Le temps d'exécution d'une structure conditionnelle est donc **la somme** des temps d'exécution des 2 branches.

⇒ **Optimisation**

- Tenter d'éviter/supprimer les branches
- S'assurer que tous les threads d'un warp prennent la même branche.

4 – Warp : Exécution

- Les différents **warps** d'un même bloc ne sont pas exécutés en parallèle.
- Pas de garantie sur l'ordre d'exécution des instructions entre threads de différents warps

⇒ **Accès concurrents à la mémoire partagée.**

- plusieurs threads de warps différents manipulant la même donnée → problème d'accès aux données

4 – Warp : Synchronisation

- Une barrière de synchronisation entre threads d'un même bloc est disponible
- Lorsqu'un **warp** arrive à la barrière, il est placé dans une liste d'attente. Une fois tous les warps arrivés à la barrière, leur exécution se poursuit après la barrière

⇒ **Structure conditionnelle**

- Dans le cas d'une structure conditionnelle, la barrière doit-être placée dans les deux branches, sinon blocage possible.

4 – Warp : Ordonnancement

- Si un **warp** doit attendre le résultat d'une longue opération (par exemple accès mémoire globale), celui-ci est placé dans un fil d'attente et un autre warp dans la liste des warps prêts à l'exécution peut-être exécuté.

- Ce mécanisme permet de masquer les opérations ayant une latence importante et d'optimiser l'utilisation des processeurs.
- **Changement de contexte** peu couteux sur GPU (comparaison CPU)

⇒ **Optimisation**

- De manière à pouvoir masquer les opérations de grande latence, il convient de placer plus de 32 threads et donc 2 warps par bloc.

4 –Blocs: placement sur les multiprocesseurs

- Chaque bloc est placé sur un multiprocesseur. Plusieurs blocs d'un même kernel peuvent s'exécuter en parallèle sur différents multiprocesseurs.

- Suivant l'architecture, des blocs de kernels différents peuvent s'exécuter simultanément sur des multiprocesseurs différents.

⇒ **Optimisation de l'occupation des multiprocesseurs**

- Sur architecture **Tesla**, le nombre de blocs doit-être au moins égal au nombre de multiprocesseurs. L'idéal étant d'avoir un multiple du nombre de multiprocesseurs.
- A partir de **Fermi**, des blocs de kernels différents peuvent s'exécuter simultanément.

4 – Bilan

- Contrôle nombre de blocs ,threads, warps.
- Visibilité politiques d’ordonnancement/placement.
- Opportunités de recouvrement calcul/transfert.

Agenda

1. Introduction et Enjeux
2. Architecture
3. Modèle de programmation
4. Modèle d’exécution
5. C/C++ API

5 – C/C++, API

2 API exclusives

- Driver : bas-niveau, verbeux
- Runtime : haut-niveau, prise en main rapide.

Runtime API

- C/C++ (11-14) + mots clés/notations
- API : CUDA Runtime API.

5 – Premiers pas …

first.cu

```
// __global__ : kernel appelable depuis host
__global__ void kernel() {}

int main()
{
   kernel<<<1, 1>>>(); // notation appel kernel
      // <<<1, 1>>> 1 bloc de 1 thread
   return 0;
}
```

Compilation : nvcc –o first first.cu

Programmation côté host

5 – Appel d'un kernel

Notation <<<n1,n2>>>

- n1 : dimensions des blocs.
- n2 : dimensions des threads.

Exemples

- <<<1, 256>>> : 1 bloc de 256 threads
- <<<256, 1>>> : 256 blocs de 1 thread chacun
- <<<16, 16>>> : 16 blocs de 16 threads chacun
- <<<dim3(4,4), dim3(4, 4)>>> : 16 blocs 2D de 16 threads 2D chacun

Dimensions limites dépendante du GPU → DeviceQuery

5 – Allocation de mémoire sur le device

cudaError_t cudaMalloc (&ptr,size)

- ptr : pointeur
- size : nombre d'octets à allouer.

cudaError_t cudaFree (ptr)

- ptr : pointeur

5 – Transfert de données **host-device**

Les transferts entre mémoires host et device de font à l'aide de la fonction :
cudaError_t cudaMemcpy (dst, src, size, dir)

- dst : pointeur vers la destination
- src : pointeur vers la source
- size : nombre d'octets à transférer
- dir : sens de la copie

  - ✓ **cudaMemcpyDeviceToHost**
  - ✓ **cudaMemcpyHostToDevice**

5 – Gestion des erreurs

Fonctions CUDA

- cudaSuccess si ok
- cudaError …. (cf. driver_types.h).

Message explicite : cudaGetErrorString (err)

Kernel

- Variable interne : cudaGetLastError

⚠ Appel asynchrone → cudaDeviceSynchronize() requis

⚠ cudaDeviceSynchronize() peut également retourner une erreur.

Programmation côté kernel

5 – Qualificateurs de kernel

__global__ : le kernel peut-être appelé à partir d'un autre kernel ou du code host
__device__ : le kernel ne peut-être appelé que par un autre kernel.

Fonctions

- Pas d'appels de fonctions « à la CPU » sur le GPU car pas de pile.
- Le code des fonctions est donc mis « inline » à la compilation

5 – Identification des threads/blocs

Variables prédéfinies

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

5 – Identification des threads/blocs

1 bloc 1D de N threads

- blockDim.x = N
- threadIdx.x

1 bloc 2D de NxM threads

- blockDim.x = N
- blockDim.y = M
- threadIdx.x
- threadIdx.y

5 – Qualificateurs de variables

Mémoire	Qualifier
registres	
shared	__shared__
local	__local__
global	__device__
constant	__constant__

Tableaux

- Les tableaux sont stockés dans la mémoire locale pas dans les registres

5 – Différents types de mémoire

Type	on-chip ?	cached ?	accès	portée	durée de vie
registres shared	oui oui	- -	rw rw	thread block	thread block
local global constant texture	non non non non	non non oui oui	rw rw r r	thread host + threads host + threads host + threads	thread gérée par le programme gérée par le programme gérée par le programme

5 – Mémoire shared

- accès aussi rapide que des registres sous certaines conditions
- quantité limitée (16kb à 48 kb)
- structurée en banques (16 banques pour Fermi / 32 pour Kepler)

Contraintes pour les performances

- Des threads distincts doivent accéder en cas d'accès simultanés à des banques distinctes
- Si tous les threads accèdent simultanément à une même banque : mécanisme de broadcast
- Si tous les threads accèdent à des données distinctes dans une même banque : sérialisation des accès.

5 – Mémoire shared

Deux méthodes pour fixer la taille

- En dur dans le kernel
- En paramètre de l'appel du kernel

Exemples

```
__global__ void kernel( ... )
{
  __shared__ float t[ 1024 ]; // en dur
}

__global__ void kernel( ... )
{
  extern __shared__ float t[]; // param.
}
...
kernel<<< grid , block , 1024 >>>( ... ); // param.
```

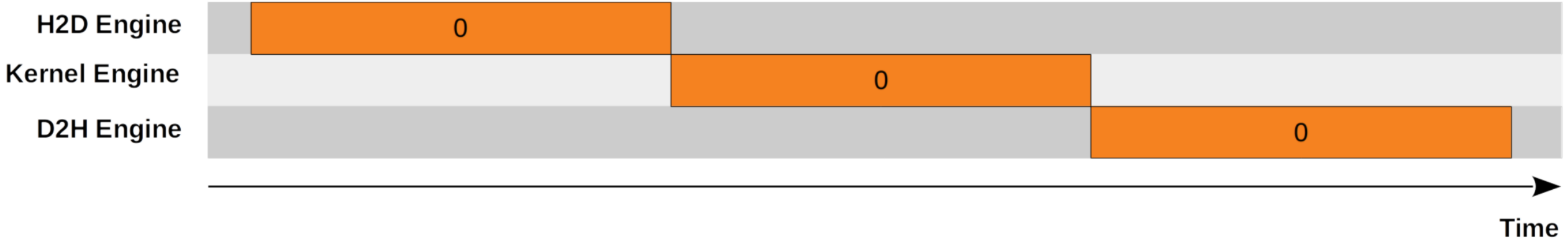
5 – Synchronisation des threads

__syncthreads ()

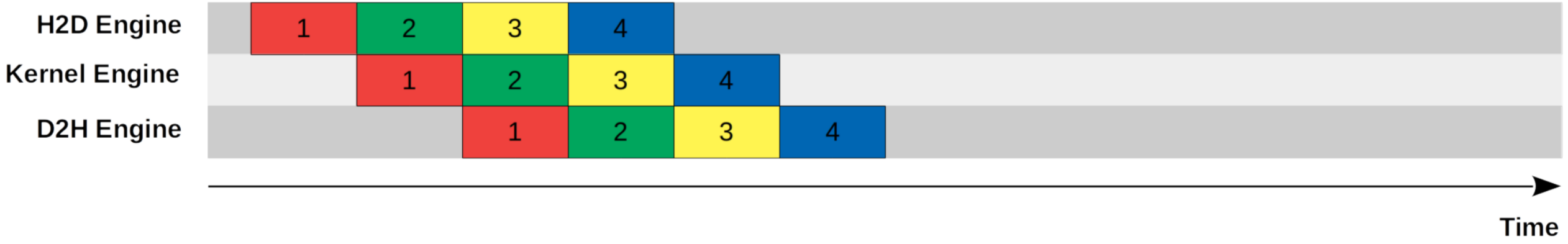
Synchronisation de tous les threads d’un bloc avec une unique primitive

5 – Streams

Serial Model



Concurrent Model



5 – Streams

- Stream : suite d'opérations exécuté de manière ordonnée
- Par défaut lancement de kernel & transfert mémoire exploite le stream 0
- Kernel dans le même stream → exécution *in-order*
- Kernel dans des streams différents → exécution *out-of-order*

- *Concurrence*

✓ *Streams différents*
✓ *Copie asynchrone sur le host*
✓ *Disponibilité des ressources*

5 – Streams

Pipeline/Recouvrement calcul-communication

```
cudaStreams_t streams [2];

cudaStreamCreate ( &streams[0] );
cudaStreamCreate ( &streams[1] );

cudaMemcpyAsync(…,streams[0] );   // non-bloquant
cudaMemcpyAsync(…,streams[1] );

kernel <<< … , streams [ 0 ] >>> ( … ) ;
kernel <<< … , streams [ 1 ] >>> ( … ) ;

cudaMemcpyAsync( …, streams[0] );
cudaMemcpyAsync( …, streams[1] );

cudaStreamDestroy( streams[0] );
cudaStreamDestroy( streams[1] );
```

5 – Events

Mesure des temps sur GPU

```
cuda Event_t start , stop ;

cudaEventCreate ( &start ) ;
cudaEventCreate ( &stop ) ;

cudaEventRecord ( start, 0 ) ;    // stream
.... // code à mesurer
cudaEventRecord ( stop , 0 ) ;

cudaEventSynchronize ( stop ) ; // attente fin événement

float duration = 0.0 f ;
cudaEventElapsedTime ( &duration, start, stop ); // ms
```

Exemples

```cpp
#include <iostream>
#include <vector>
#include <algorithm>


__global__ void vecadd( int * v0, int * v1, std::size_t size )
{
  auto tid = threadIdx.x;
  v0[ tid ] += v1[ tid ];
}
```

```cpp
int main()
{
  std::vector< int > v0( 100 );
  std::vector< int > v1( 100 );

  int * v0_d = nullptr;
  int * v1_d = nullptr;

  for( std::size_t i = 0 ; i < v0.size() ; ++i )
  {
    v0[ i ] = v1[ i ] = i;
  }

  cudaMalloc( &v0_d, v0.size() * sizeof( int ) );
  cudaMalloc( &v1_d, v1.size() * sizeof( int ) );

  cudaMemcpy( v0_d, v0.data(), v0.size() * sizeof( int ), cudaMemcpyHostToDevice );
  cudaMemcpy( v1_d, v1.data(), v1.size() * sizeof( int ), cudaMemcpyHostToDevice );

  vecadd<<< 1, 100 >>>( v0_d, v1_d, v0.size() );

  cudaMemcpy( v0.data(), v0_d, v0.size() * sizeof( int ), cudaMemcpyDeviceToHost );

  for( auto const x: v0 )
  {
    std::cout << x << std::endl;
  }

  cudaFree( v0_d );
  cudaFree( v1_d );

  return 0;
}
```

Bilan

Programmation simple…

- CUDA = C/C++ avec quelques mots-clés
- compilation aisée

Optimisation parfois complexe !

- optimisations classiques : déroulage de boucles
- dépend de l’architecture GPU : nombres de générations + variantes
- différents types de mémoire
- répartition des threads en blocs, grilles + warps
- synchronisation
- recouvrement calculs/communications

Fin