NVIDIA CUDA

Compute **U**nified **D**evice **A**rchitecture

Sylvain Jubertie
sylvain.jubertie@univ-orleans.fr

NVIDIA CUDA

Compute Unified Device Architecture

1. cartes graphiques Nvidia
2. + pilotes CUDA
3. + extension langage C/C++
4. + compilateurs, outils, libs

Installation : Logiciels

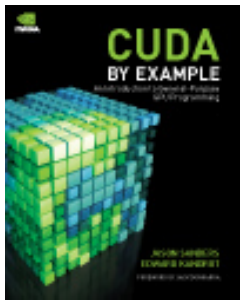
Téléchargement sur le site
`https://developer.nvidia.com/cuda-downloads`

1. Pilote Nvidia
2. CUDA Toolkit : compilateur, debugger, documentation, exemples
3. Nvidia NSight (Visual Studio ou Eclipse)
4. Bibliothèques : CuBLAS, CuFFT, NPP, ...

Ressources : Livres

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

- Jason Sanders et Edward Kandrot, *CUDA by Example*



Ressources : Internet

- Nvidia CUDA Zone
- Dr Dobbs - CUDA, Supercomputing for the Masses

Gammes Nvidia

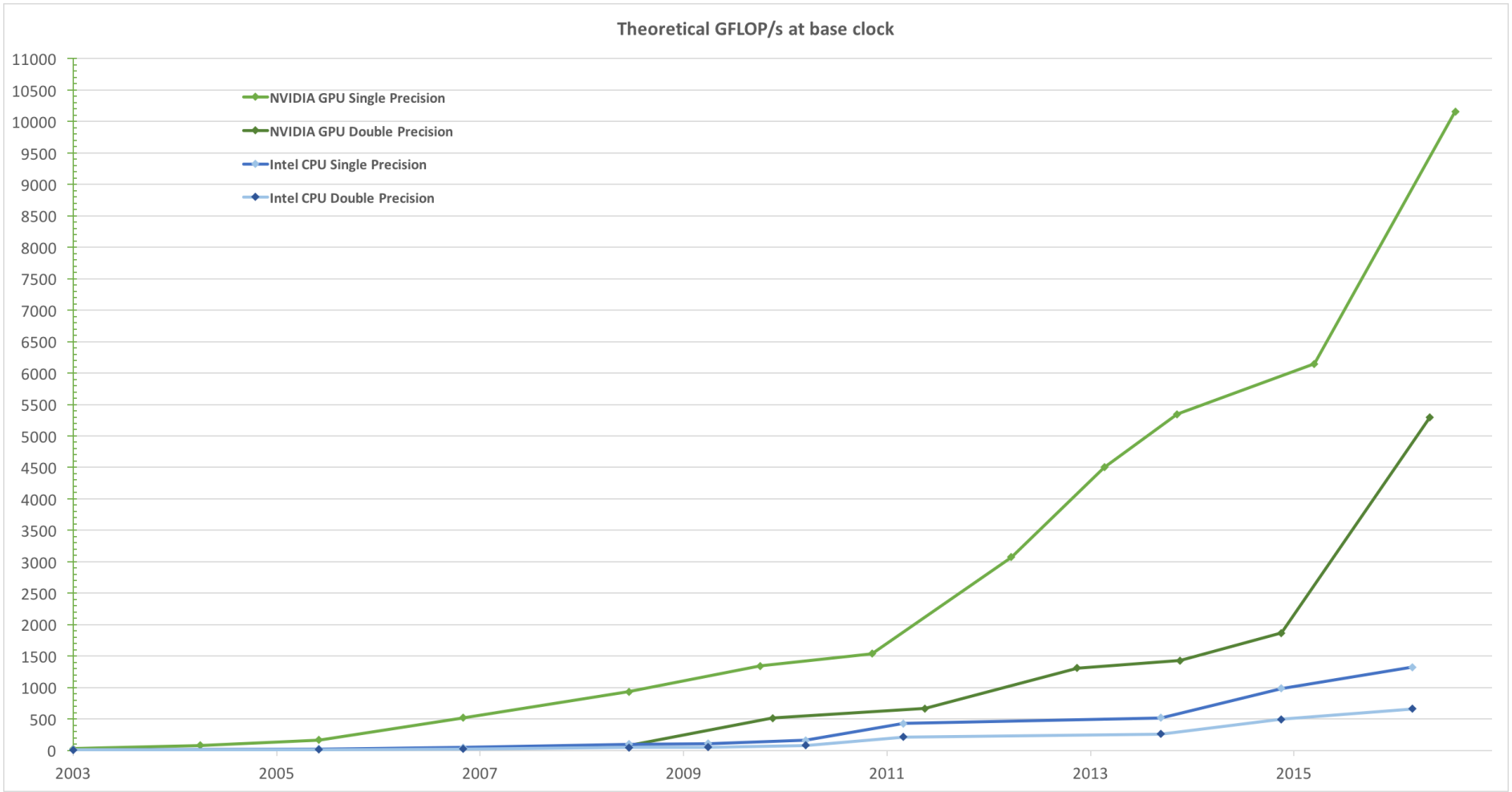
- GeForce : gamme grand public (jeu)
- Quadro : gamme professionnelle pour la 3D : 3D stéréo (quad-buffer).
- Tesla : Gamme professionnelle pour le GPGPU : pas/moins de composants vidéo, mémoire ECC.
- Tegra : gamme pour les plateformes nomades (tablettes, smartphones, voitures autonomes) : processeur ARM multicoeurs + GPU Nvidia présent dans les Nvidia Shields, la Nintendo Switch

Pourquoi utiliser des GPU ?

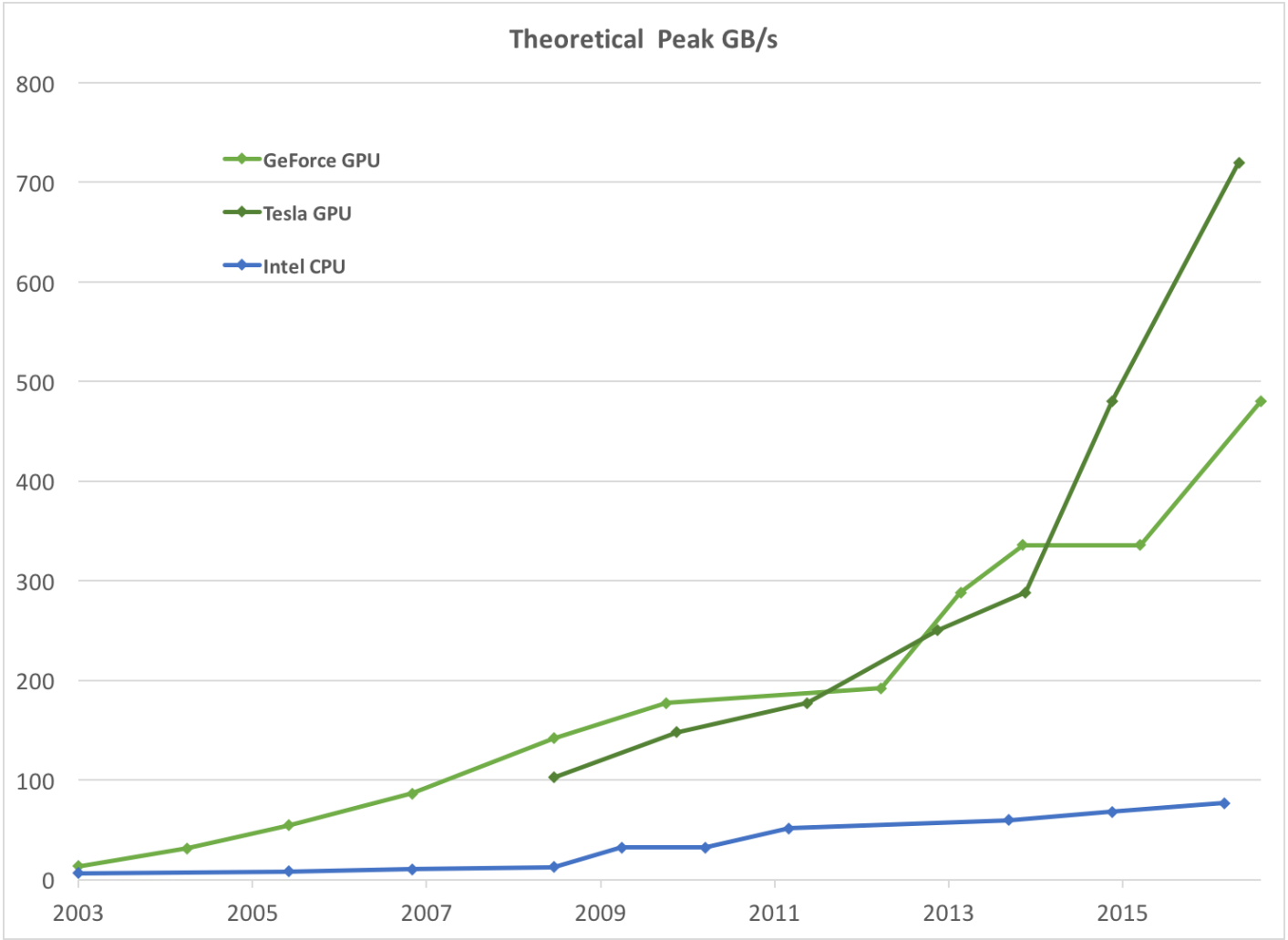
1. performances : architecture many-core
2. rapport performances / énergie
3. rapport performances / prix

	Intel Xeon E5-2699 v4	Nvidia GTX 1080
coeurs	22	2560
fréq.	2.2 Ghz(boost 3.6 Ghz)	1.607 Ghz(boost 1.733 Ghz)
mémoire	76.8 GB/s	320 GB/s
conso.	145W	180W
prix	4115$	700€

Sources : Intel ARK, Nvidia website



Puissance de calcul supérieure à celle des CPU.



Bande passante supérieure à celle des CPU.

Questions à se poser

Mais... il faut que le code se prête à la parallélisation sur GPU :

- Calcul simple ou double précision ?
- Parallélisme de données ?
- Taille des données ? 4 à 24GB “seulement” sur GPU.
- ...

Gains à espérer (d'après Nvidia, articles, ...)

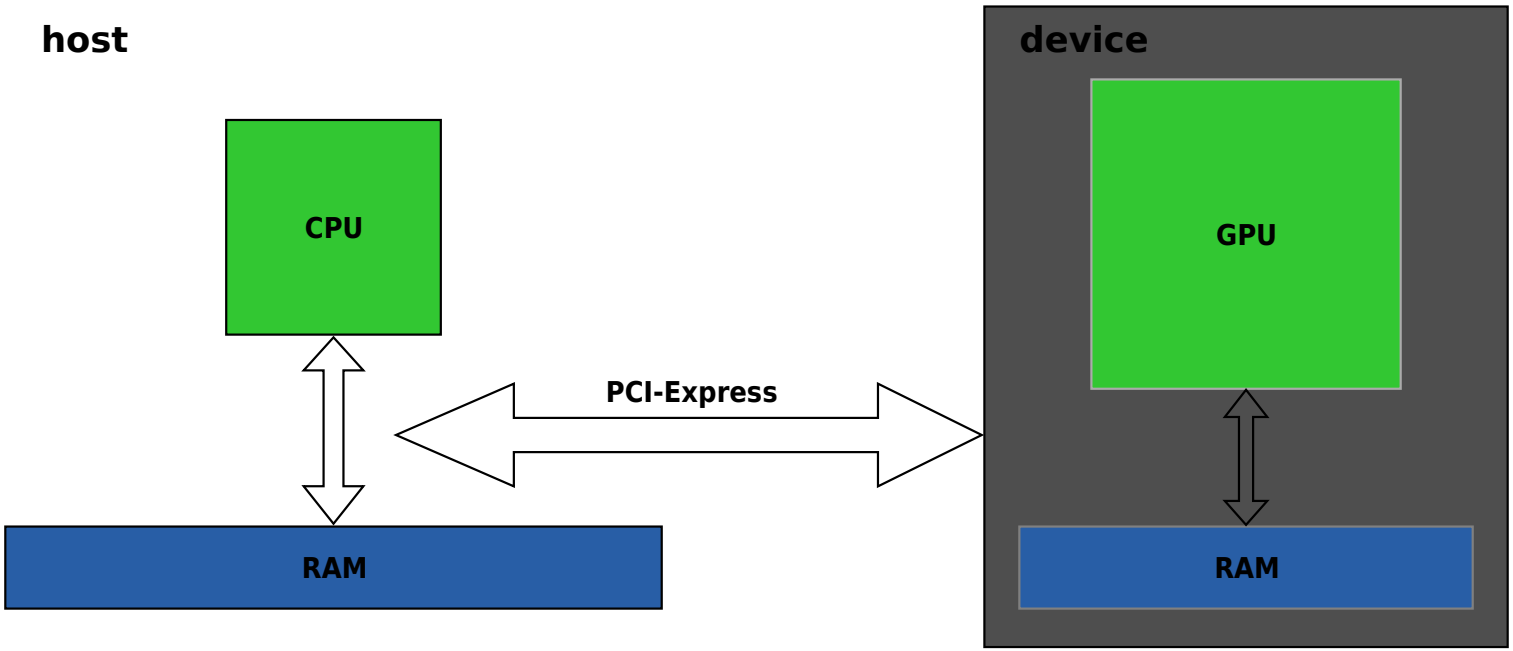
- Dynamique moléculaire : VMD (visualisation) x100, Gromacs x2-5x, NAMD x2-7, FastROCS (similarité, comparaison) x800-3000
- Finance : SciComp x10-35, Murex x60-400
- Imagerie médicale : x20-100
- Exploration sismique : x4-20
- Mécanique des fluides : x2-20

Gains à relativiser !

Attention aux comparatifs

- comparaison à des codes CPU optimisés (caches, unités SIMD, pipeline, ...) ?
- comparaison à des codes exécutés sur multi-coeurs ?
- prise en compte des communications entre CPU et GPU ?

Architecture générale



Générations de processeurs GPU

- Tesla (2008)
- Fermi (2010)
- Kepler (2012)
- Maxwell (2014)
- Pascal (2016)
- Volta (2017)
- Turing (2018)

Variantes

Chaque génération de GPU possède des variantes :

- Fermi : GF100, GF104, GF106, GF108
- Kepler : GK104, GK110
- Maxwell : GM104

Chaque nouvelle variante apporte de nouvelles capacités :

- calculs double précision
- accès aux mémoires
- fonctions atomiques
- ...

La liste de ces **computes capabilities** est disponible dans le *Nvidia CUDA C Programming Guide*.

Mémoires

1. dans le GPU : mémoires rapides mais de tailles limitées
   - registres
   - mémoire partagée, gérée par le développeur
   - caches : constantes et textures
2. dans la DRAM : mémoire plus lente mais en grande quantité
   - mémoire locale et globale
   - constantes
   - textures

Différents types de mémoire

Type	on-chip?	cached?	accès	performance
registers	oui	-	rw	1 cycle
shared	oui	-	rw	1 cycle
local	non	non	rw	lent
global	non	non	rw	lent
constant	non	oui	r	1...10...100
texture	non	oui	r	1...10...100

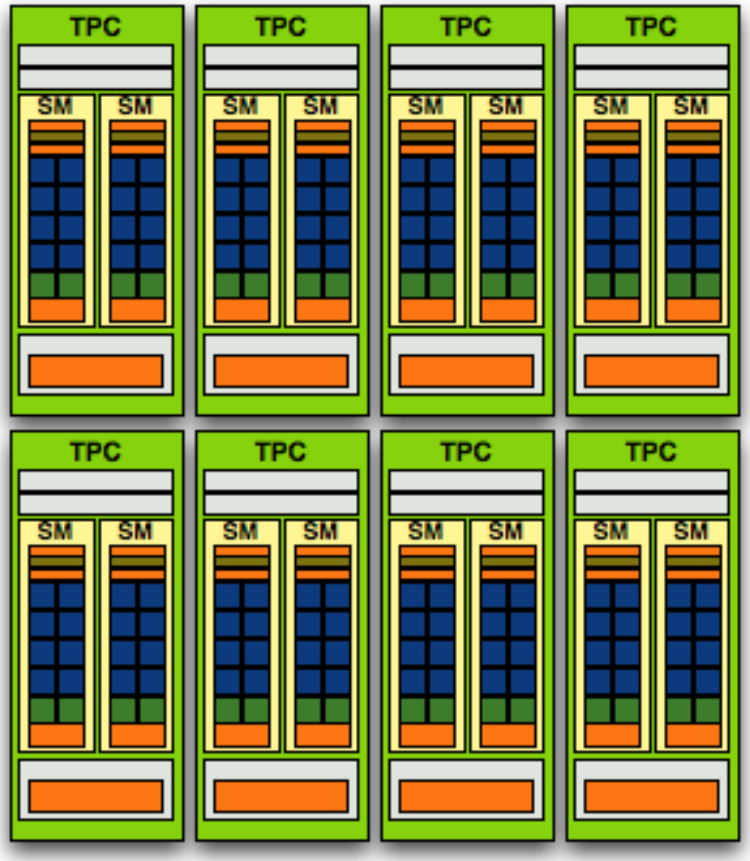
Architecture Tesla

Les processeurs sont regroupés par 8 pour former des multi-processeurs.
Les multi-processeurs sont eux-mêmes regroupés par 2 (G80) ou 3 (GT200) au sein de **Texture Processing Clusters (TPC)**.

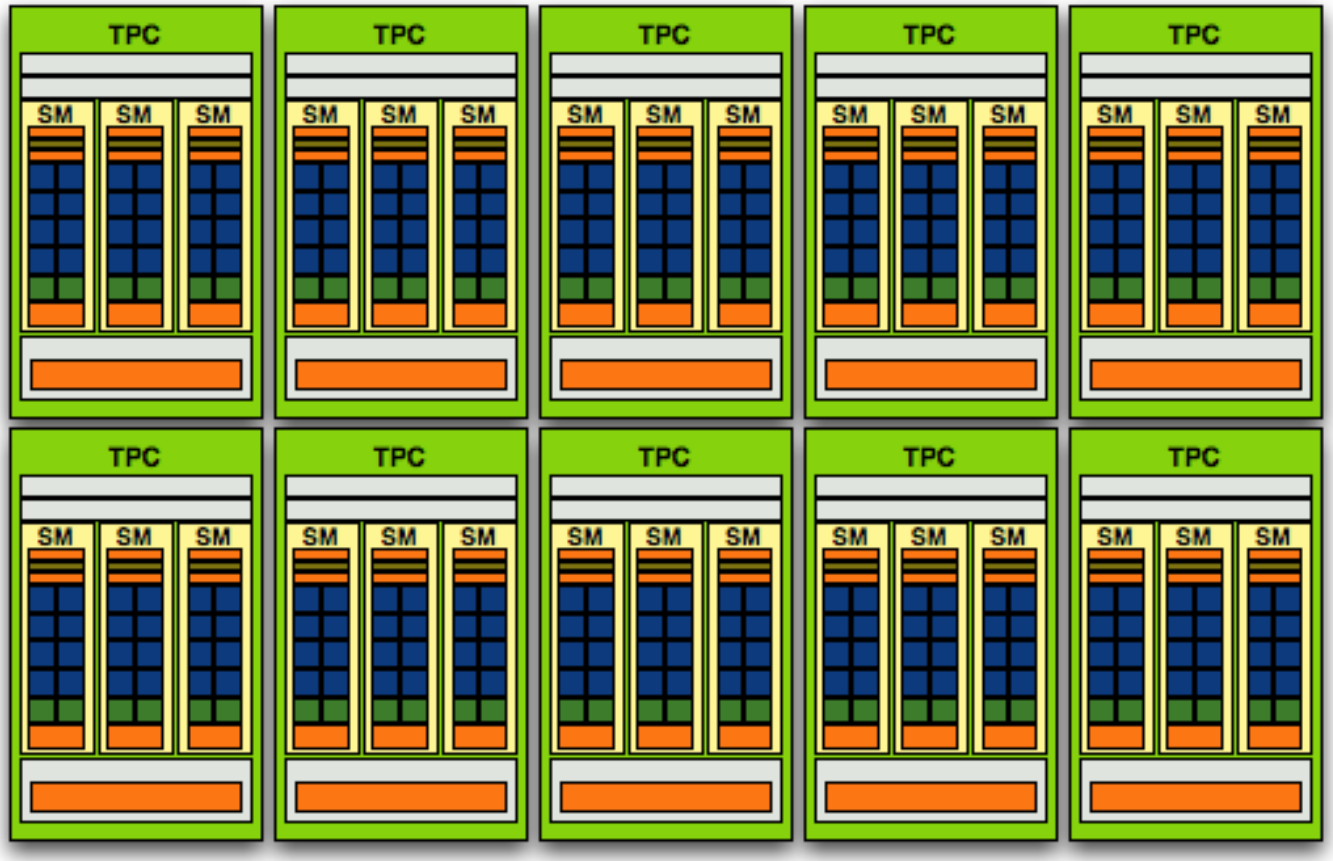
Exemple : Geforce GTX260

216 processeurs décomposés en 27 multi-processeurs.

Architecture Tesla (G80)



Architecture Tesla (GT200)



Architecture Tesla - Mémoires

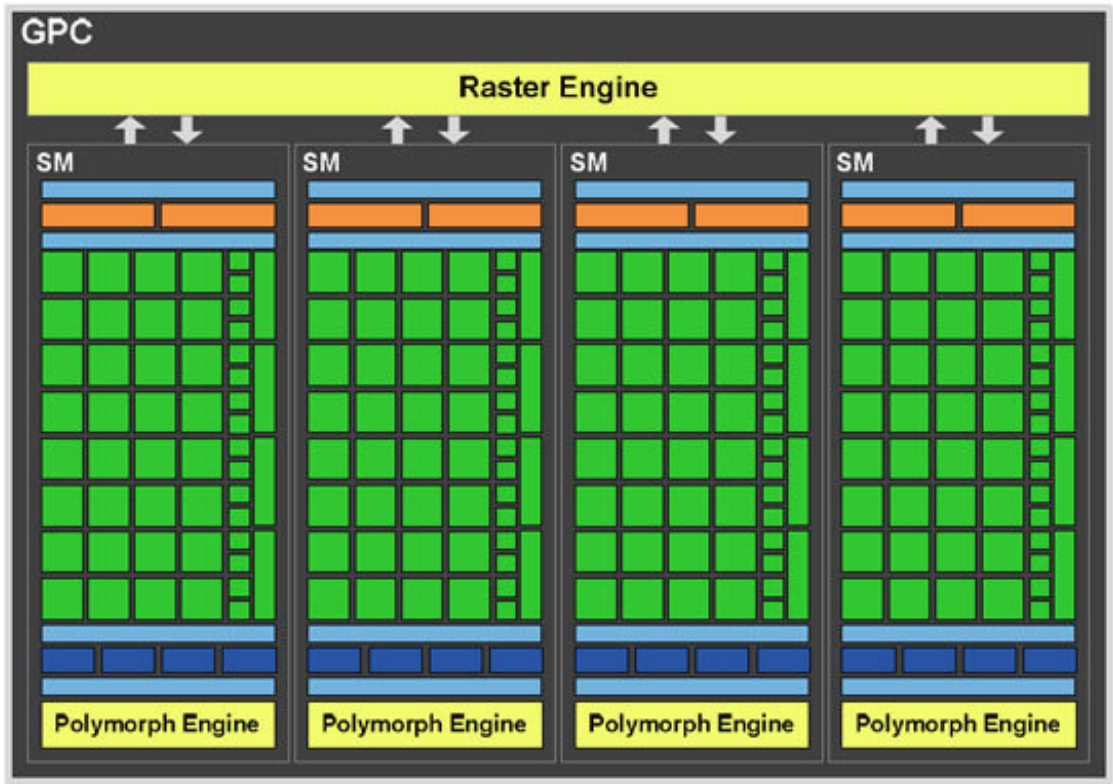
1. registres
2. mémoire partagée : par multi-processeur 16 KB divisés en 16 banques
3. mémoire globale
4. mémoire constante : 64 KB

Architecture Fermi

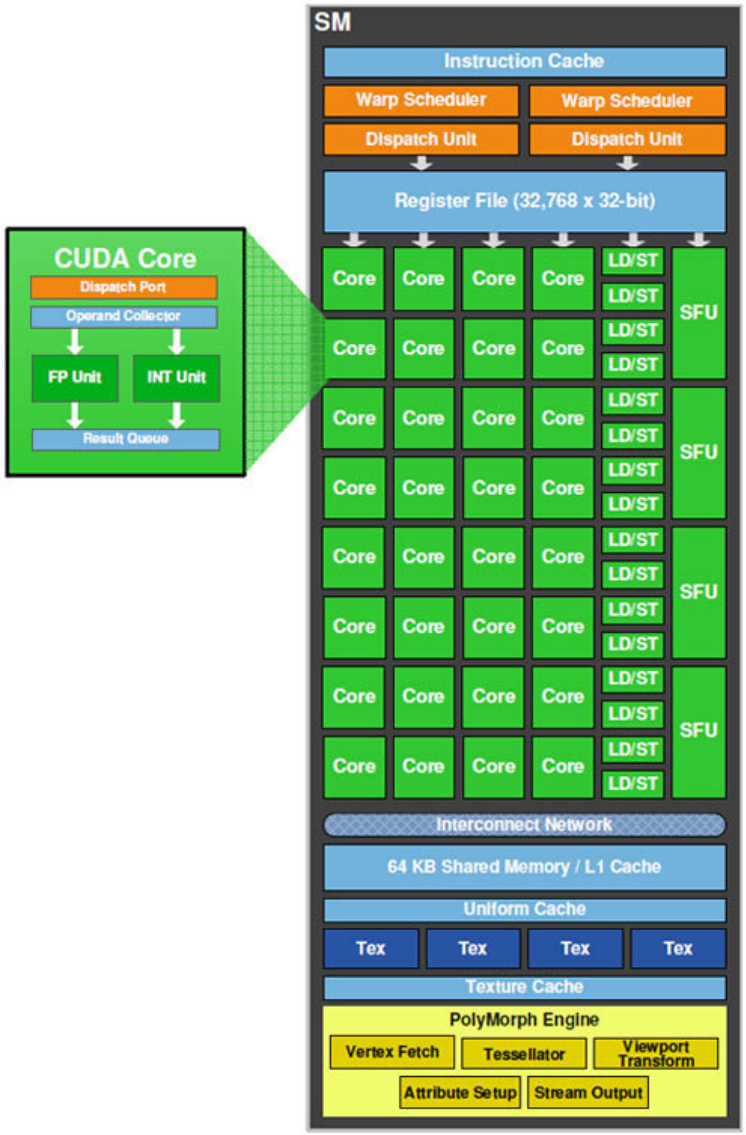
Les processeurs sont regroupés par 32 pour former des multi-processeurs.
Un GPU doté de 512 processeurs est donc composé de 16 multi-processeurs.
Les multi-processeurs sont regroupés par 4 au sein de **Graphics Processing Clusters**.

Architecture Fermi

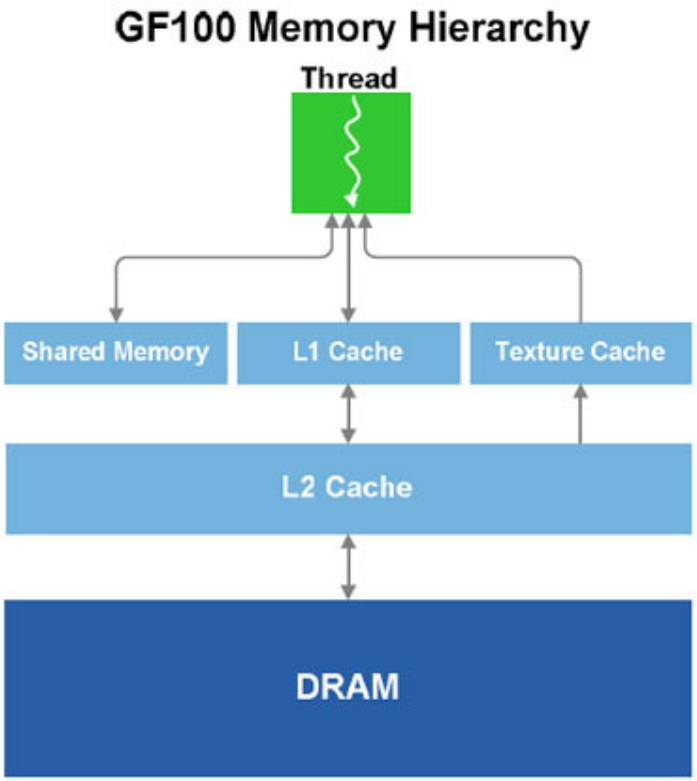
Architecture Fermi - GPC



Architecture Fermi - Multiprocesseur



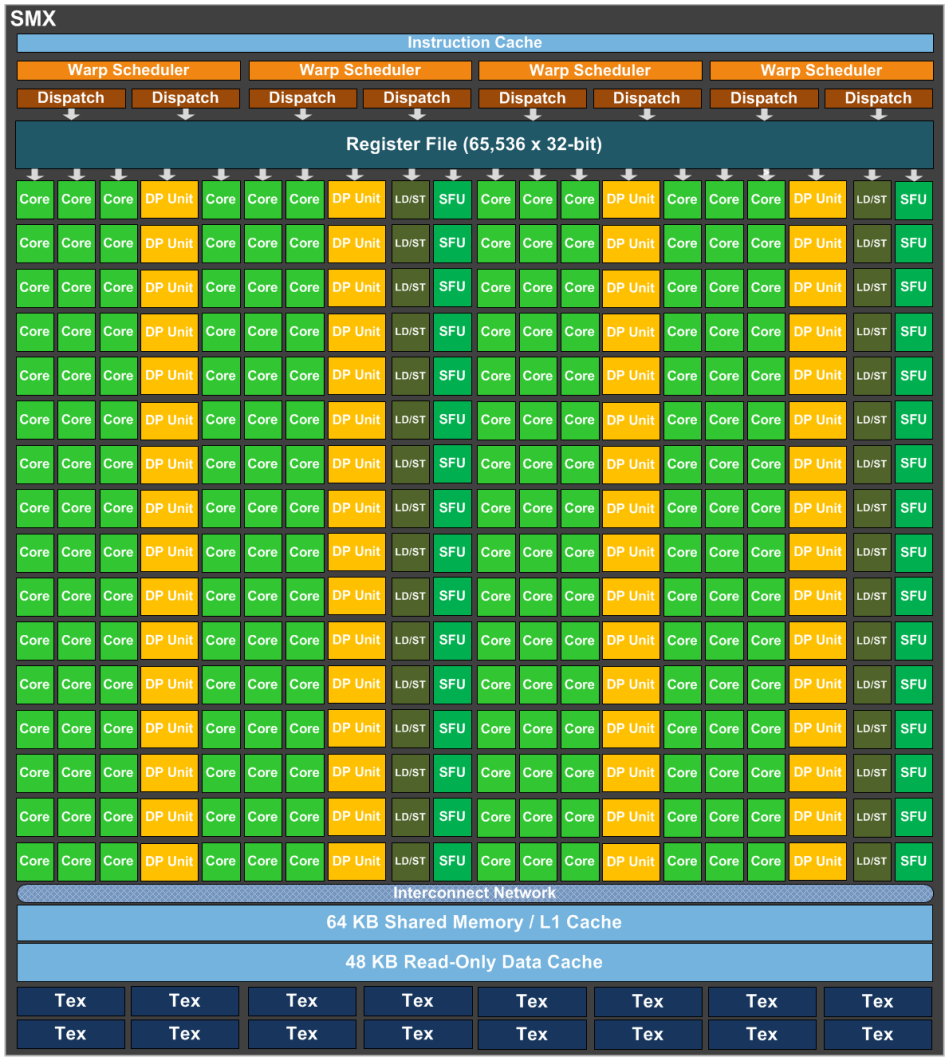
Architecture Fermi - Mémoires



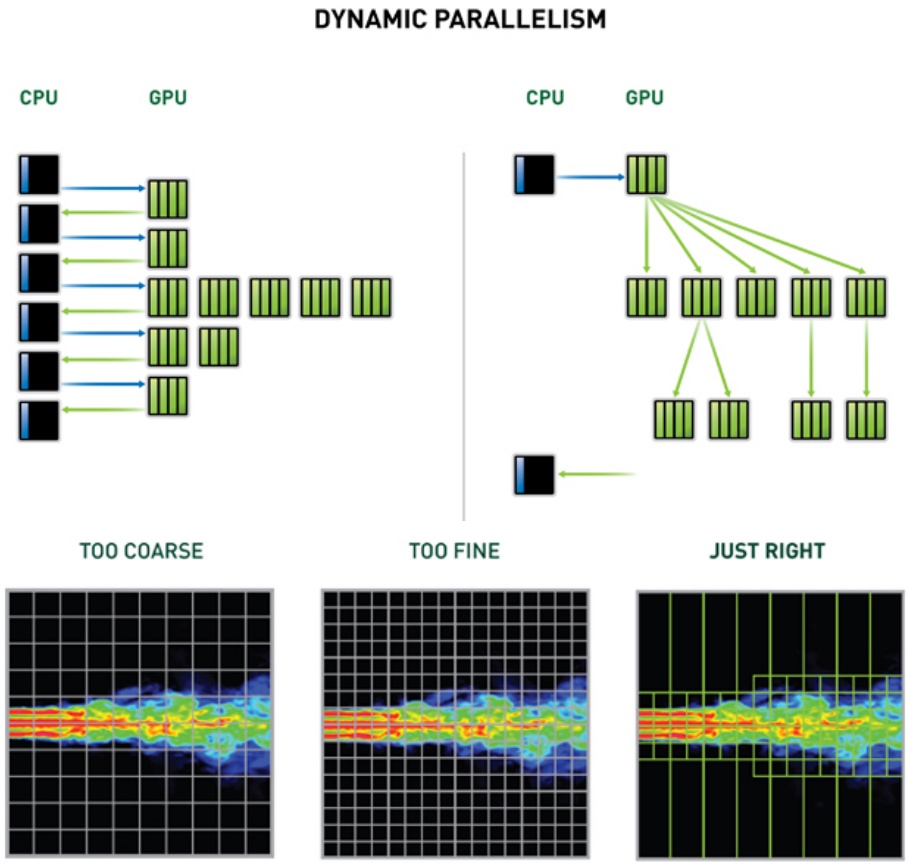
Architecture Kepler

- 1536/2880 coeurs
- 3x performance/watt par rapport à l'architecture Fermi
- parallélisme dynamique (uniquement GK110)
- GPUDirect : communication directement de carte à carte à travers le réseau.
- ...

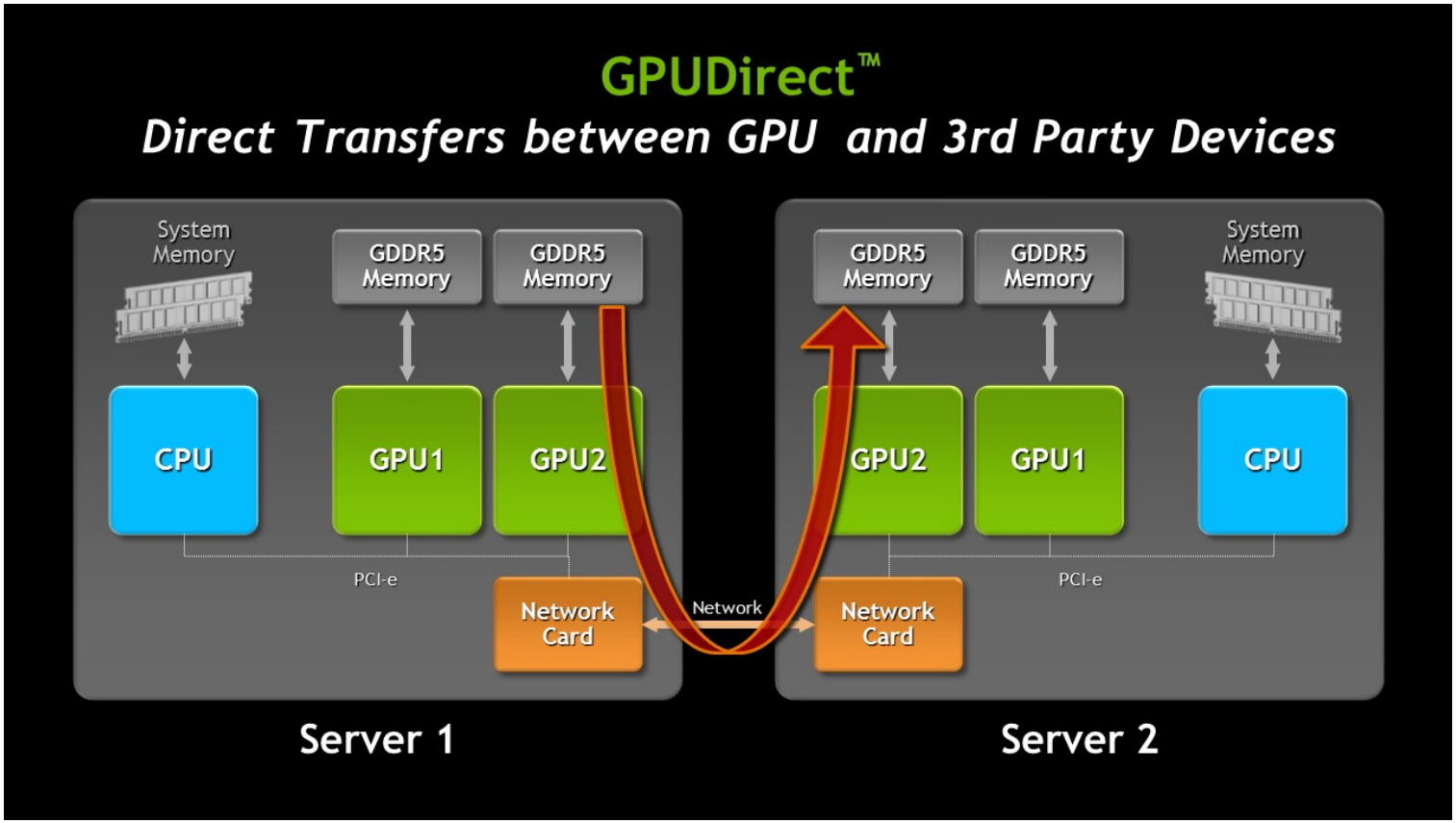
Architecture Kepler - SMX



Architecture Kepler - Parallélisme dynamique

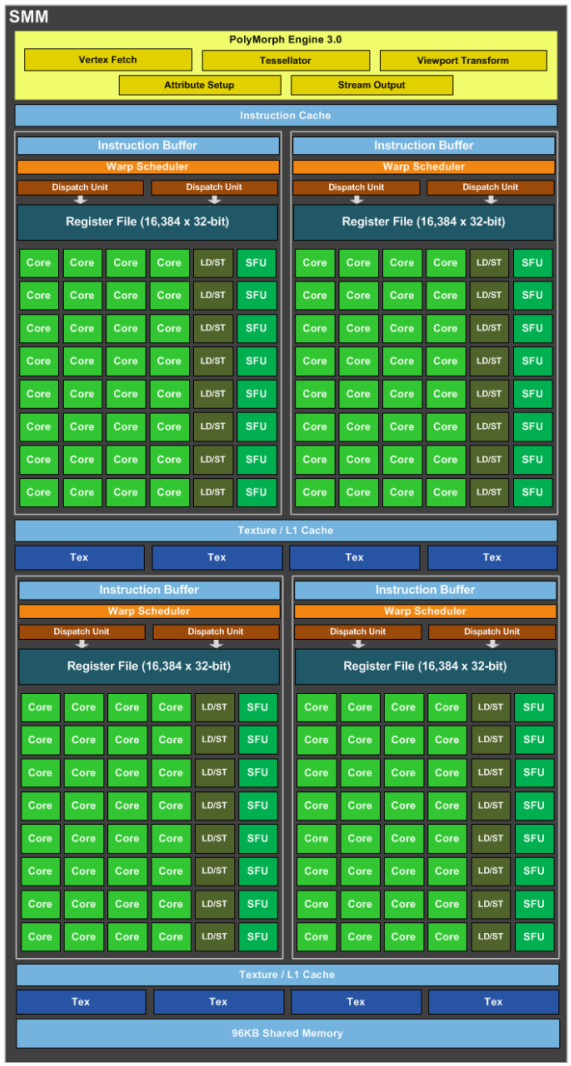


Architecture Kepler - GPUDirect

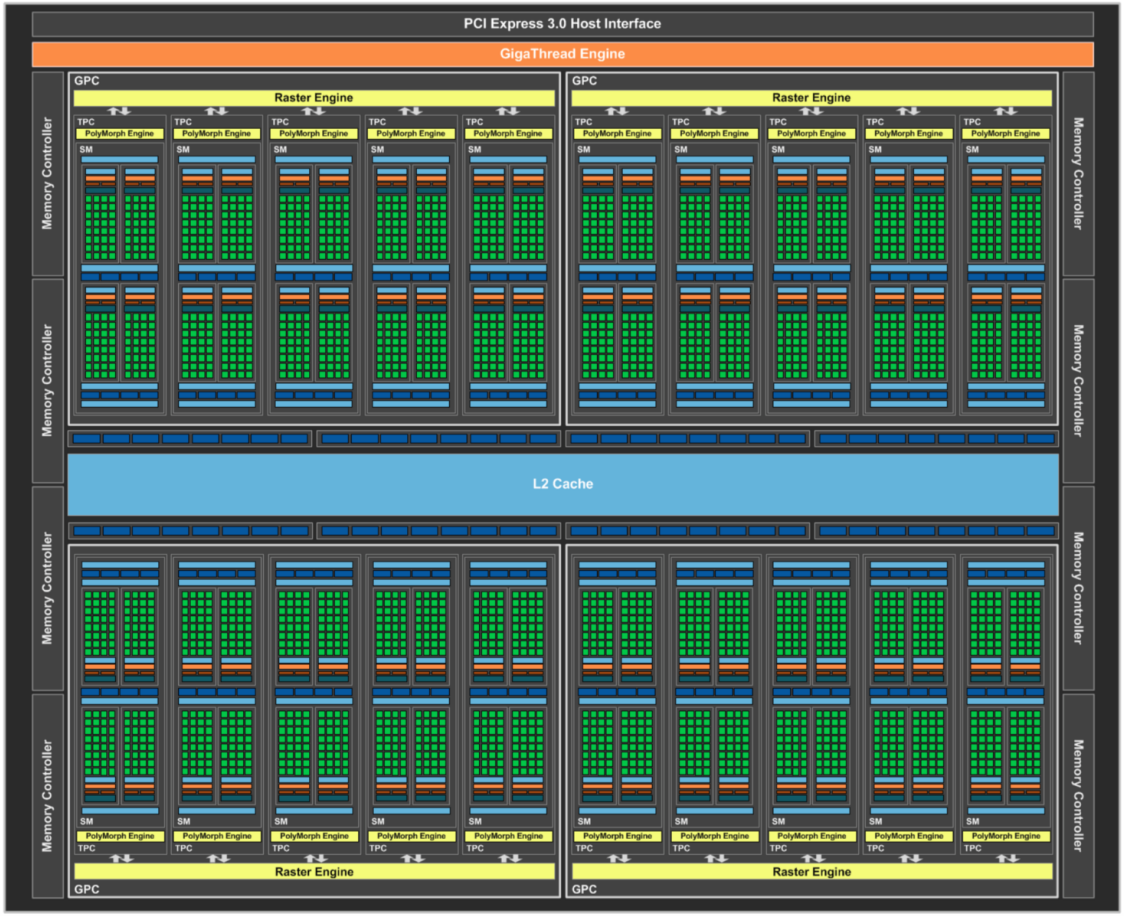


Architecture Maxwell

Architecture Maxwell



Architecture Pascal



2560 coeurs

Architecture Pascal

Divergence entre gamme joueur et gamme HPC : double precision, mémoire HBM

Démo

Pour obtenir les caractéristiques d'une carte : outil `deviceQuery` dans les exemples du SDK

```
cd dossierexemples/1_Utilities/deviceQuery
cd ../../bin/x86_64/linux/release/
./deviceQuery
```

Architectures : comparatif

Gamme Tesla : du Kepler au Volta

- nb coeurs x2
- fréquence x2
- bande passante mémoire x3

Tesla Model	K10	K20	K20X	K40	K80	M4	M40	M6	M60	P4	P40	P100	P100	P100	V100	V100	V100
Bus	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	PCI-E	SXM2	HGX-1	PCI-E	SXM2
GPU	2 * GK104	GK110	GK110	GK110B	2 * GK210B	GM206	GM200	GM204	2 * GM204	GP104	GP102	GP100	GP100	GP100	GV100	GV100	GV100
FP32 Cores	2 * 1,536	2,496	2,688	2,880	4,992	1,024	3,072	1,536	4,096	2,560	3,840	3,584	3,584	3,584	5,120	5,120	5,120
FP64 Cores	-	-	-	-	-	-	-	-	-	640	960	1792	1792	1792	2,560	2,560	2,560
Tensor Cores	-	-	-	-	-	-	-	-	-	-	-	-	-	-	640	640	640
Base Core Clock Speed	745 MHz	706 MHz	732 MHz	745 MHz	560 MHz	872 MHz	948 MHz	950 MHz	899 MHz	810 MHz	1,303 MHz	1,126 MHz	1,126 MHz	1,328 MHz	***823 MHz***	***1,097 MHz***	***1,372 MHz***
GPU Boost Clock Speed	-	-	-	875 MHz	875 MHz	1,072 MHz	1,114 MHz	1,051 MHz	1,177 MHz	1,063 MHz	1,531 MHz	1,303 MHz	1,303 MHz	1,480 MHz	***918 MHz***	***1,224 MHz***	1,530 MHz
SMXs or SMMs	2 * 8	13	14	15	2 * 13	8	24	12	2 * 16	20	30	56	56	56	80	80	80
Base Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*
Peak Tensor Core, Teraops	-	-	-	-	-	-	-	-	-	-	-	-	-	-	***100.0***	112.0	125.0
Base INT8, Teraops	-	-	-	-	-	-	-	-	-	16.6	40	-	-	-	-	-	-
Peak INT8, Teraops	-	-	-	-	-	-	-	-	-	21.8	47	-	-	-	***50.2***	56.0	62.8
Base HP, Teraflops	-	-	-	-	-	-	-	-	-	-	-	*	*	*	*	*	*
Peak HP, Teraflops	-	-	-	-	-	-	-	-	-	-	-	18.7	18.7	21.2	***25.1***	28.0	31.4
Base SP, Teraflops	4.58	3.52	3.95	4.29	5.6	*	*	*	*	*	*	*	*	*	*	*	*
Peak SP, Teraflops	4.58	3.52	3.95	5.0	8.74	2.2	7.0	3.2	9.6	5.45	11.8	9.3	9.3	10.6	***12.6***	14.0	15.7
Base DP, Teraflops	0.19	1.17	1.31	1.43	1.87	*	*	*	*	*	*	*	*	*	*	*	*
Peak DP, Teraflops	0.19	1.17	1.31	1.66	2.91	0.06	0.20	*	0.30	0.17	0.37	4.70	4.70	5.30	***6.2***	7.00	7.80
GDDR5/HBM2 Memory	8 GB	5 GB	6 GB	12 GB	24 GB	4 GB	24 GB	8 GB	16 GB	8 GB	24 GB	12 GB	16 GB	16 GB	16 GB	16 GB	16 GB
Memory Clock Speed	2.5 GHz	2.6 GHz	2.6 GHz	3.0 GHz	2.5 GHz	2.75 GHz	3.0 GHz	2.3 GHz	2.51 GHz	3.0 GHz	3.6 GHz	-	-	-	-	-	-
Memory Bandwidth	320 GB/sec	208 GB/sec	250 GB/sec	288 GB/sec	480 GB/sec	88 GB/sec	288 GB/sec	147 GB/sec	320 GB/sec	192 GB/sec	346 GB/sec	540 GB/sec	720 GB/sec	720 GB/sec	900 GB/sec	900 GB/sec	900 GB/sec
Power Draw	225 W	225 W	235 W	235 W	300 W	50 W – 75 W	250 W	100 W	250/300 W	50/75 W	250 W	250 W	250 W	300 W	150 W	250 W	300 W

Pour l'instant...

- plusieurs générations d'architectures différentes : Tesla, Fermi, Kepler, ..., Turing
- codes CUDA (presque) portables d'une architecture à l'autre mais...
- optimisations spécifiques à chaque architecture !
- **performances presque portables !**
- nécessite une connaissance approfondie des architectures et de leur fonctionnement.
- mêmes problèmes pour OpenCL.

Modèles

- Les 2 sections suivantes sont les plus délicates à appréhender et contiennent beaucoup de texte pour expliquer précisement les modèles.
- Comprendre les modèles est nécessaire pour exploiter les performances des GPU, sinon autant ou moins de performance qu'un CPU bien exploité.
- Ne pas hésiter à m'interrompre !

Programmation hétérogène

Un programme contient à la fois :

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

Code **host**

Le code **host** contrôle l'exécution du code **device** ainsi que les communications entre la mémoire **host** et **device**.

Kernel

Un **kernel** est une procédure (pas de valeur de retour) destinée à être exécutée par les coeurs du GPU.
Un kernel peut appeler un autre kernel.
Un kernel peut être appelable par l'hôte ou uniquement par un autre kernel.

Côté **device** : Modèle SPMD

Data parallélisme

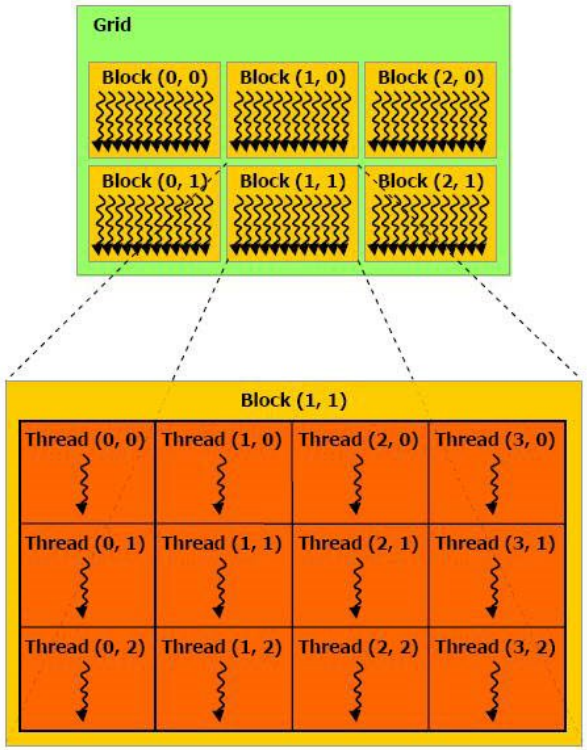
Modèle SPMD : Single Program on Multiple Data.
À l'exécution le code host va instancier un kernel avec une configuration/topologie 1D, 2D ou 3D.

Thread

- Une instance de kernel est appelée **thread**.
- Chaque **thread** possède des identifiants propres 1D, 2D ou 3D, suivant la topologie, permettant de les distinguer.
- 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.

Organisation des threads

Les threads peuvent être regroupés en blocs, eux-mêmes regroupés en une grille, chacun possédant un identifiant propre (1D, 2D ou 3D), on parle alors des coordonnées d'un thread ou d'un bloc.



Remarques

Choix du nombre de threads, de la topologie :

- Sur CPU : nombre de threads correspondant au nombre de coeurs disponibles
- Sur GPU : nombre de threads correspondant au nombre de données à traiter

Exemples

- Somme de 2 vecteurs 1D de n flottants $\Rightarrow$ n threads, topologie 1D
- Traitement d'une image de $x \times y$ pixels $\Rightarrow$ blocs 2D (bx, by) de threads, nombre de threads $= (x/bx).bx.(y/by).by$, soit éventuellement plus de threads que de pixels à traiter !

Déroulement d'un programme CUDA

1. initialisation des mémoires
   - initialisation des données en mémoire **host**
   - allocation dans la mémoire globale **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
4.
   - copie des résultats en mémoire **device** vers la mémoire **host** `cudaMemcpy`
   - exploitation directe des résultats par OpenGL pour affichage
5. libération de la mémoire globale **device** `cudaFree`

Compilation

Le code **host** et **device** peuvent être dans le même fichier portant l'extension `.cu`. Le compilateur `nvcc` sépare les codes **host** et **device** et les compile séparément :

- 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

Code **host** : Appels des fonctions CUDA

- Appels CUDA asynchrones, mais la dépendance des appels est respectée
- Chaque appel est envoyé au driver et mis en file d'attente s'il est dépendant d'un appel en cours de traitement
- Il est possible d'effectuer un transfert de données (upload ou download) et l'exécution d'un kernel en parallèle s'ils sont indépendants
  $\Rightarrow$ recouvrement calcul/communication
- Si le device possède 2 **copy engines** (deviceQuery) il est possible d'effectuer 1 upload, 1 download et l'exécution d'un kernel indépendant en même temps
- Attention : Ne pas mesurer le temps d'exécution des appels CUDA à l'aide des méthodes habituelles : `gettimeofday`, `std::chrono`, ...
  $\Rightarrow$ CUDAEvents

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 coeurs.

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

Règles

1. Les threads d'un même bloc sont exécutés sur un même multiprocesseur. Chaque multiprocesseur possédant une mémoire partagée, cela permet aux threads d'un même bloc de communiquer par cette mémoire.
2. Un multiprocesseur peut se voir attribuer plusieurs blocs suivant les ressources disponibles (registres).
3. Le nombre de threads par bloc est limité. Cette limite dépend de l'architecture (Tesla, Fermi).
4. Les threads d'un même bloc sont exécutés instruction par instruction par groupe de 32 threads consécutifs : un **warp.**

Warps sur l'architecture Tesla

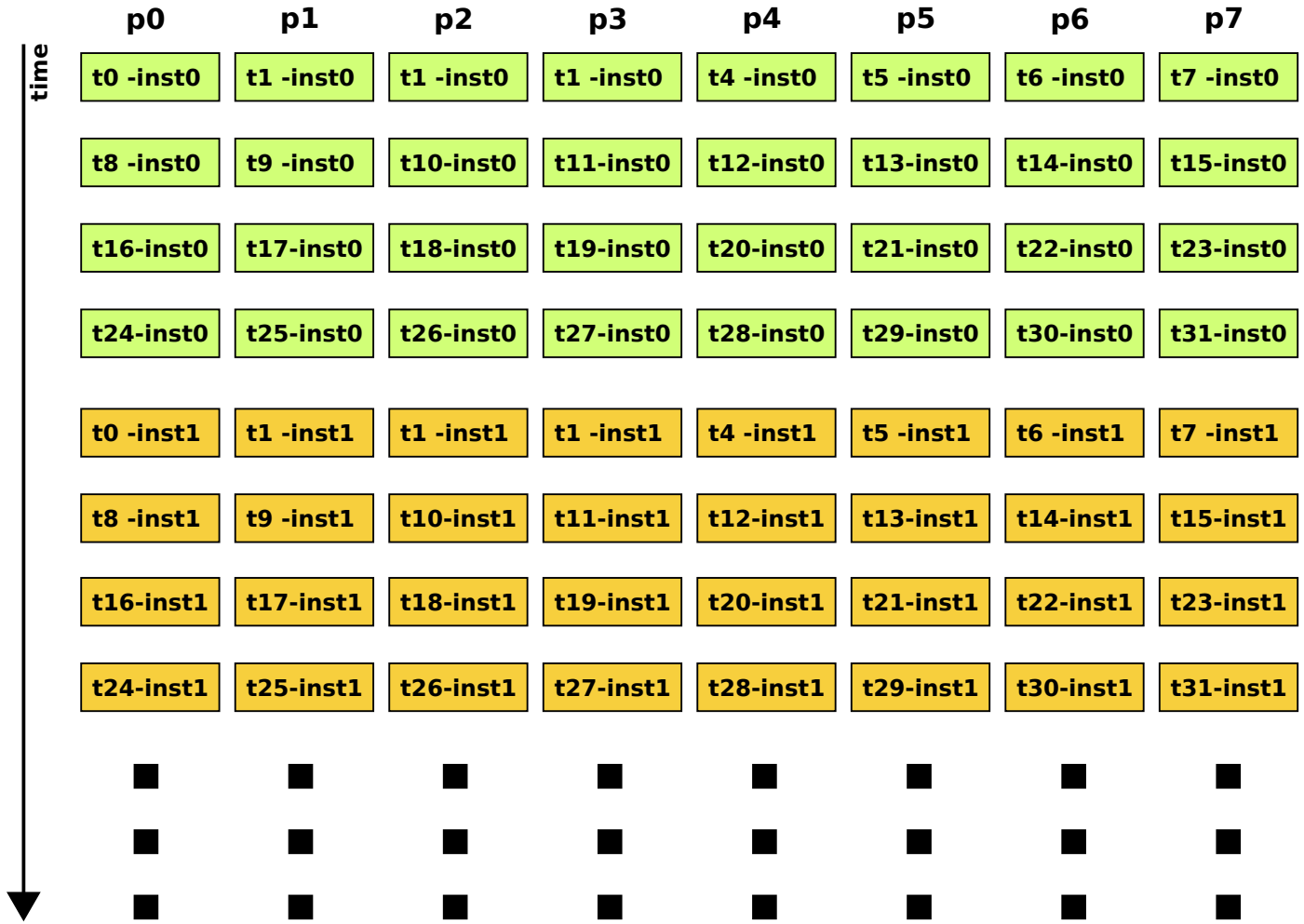
- Les threads d'un même warp sont exécutés instruction par instruction : sur le Tesla, le multiprocesseur (8 processeurs sur Tesla) exécute la première instruction du kernel sur les 8 premiers threads simultanément puis passe au 8 suivants ...
- Une fois l'instruction exécutée sur les 32 threads, on recommence avec l'instruction suivante jusqu'à la fin du kernel. Un multiprocesseur exécute donc les instructions des threads suivant le modèle SIMT : Single Instruction Multiple Threads.

Heuristique d'optimisation

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

Warps sur l'architecture Tesla

Exécution des threads d'un warp sur multiprocesseur d'architecture Tesla



Warps : 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 SIMT force l'évaluation séquentielle des 2 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

1. Essayer de supprimer les branches
2. S'assurer que tous les threads d'un warp prennent la même branche

Warps : Exécution

- Les différents warps d'un même bloc ne sont pas exécutés en parallèle.
- Il n'y a aucune garantie sur l'ordre d'exécution des instructions entre threads de différents warps.

Accès concurrents à la mémoire partagée

Il peut y avoir des problèmes d'accès concurrents aux données en mémoire partagée si 2 threads de 2 warps différents manipulent la même donnée.

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.

Warps : Scheduling

- Si un warp doit attendre le résultat d'une longue opération (par exemple accès mémoire globale), celui-ci est placé dans une file 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.

Heuristique d'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.

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.
- À partir de **Fermi**, des blocs de kernels différents peuvent s'exécuter simultanément.

C/C++, API

2 API exclusives :

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

Runtime API :

- C/C++(11-14) + mots clés/notations
- ne supporte pas les dernières versions de GCC
- API : CUDA Runtime API

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`

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 dépendantes du GPU → `deviceQuery`

Allocation de mémoire sur le **device**

```
cudaError_t cudaMalloc(&ptr, size)
```

1. `ptr` : pointeur
2. `size` : nombre d'octets à allouer

```
cudaError_t cudaFree(ptr)
```

1. `ptr` : pointeur

Transfers de données **host - device**

Les transfers entre mémoires host et device se font à l'aide de la fonction :

```
cudaError_t cudaMemcpy(dst, src, size, dir)
```

1. `dst` : pointeur vers la destination
2. `src` : pointeur vers la source
3. size : nombre d'octets à transférer
4. dir : sens de la copie
   - cudaMemcpyDeviceToHost
   - cudaMemcpyHostToDevice

Gestion des erreurs

Fonctions CUDA

→ valeur de type `cudaError_t` :

- cudaSuccess si ok
- cudaError... (dans `driver_types.h`) sinon

Message explicite : `cudaGetErrorString(err)`

Kernel

→ variable interne : `cudaGetLastError()` (ou `cudaPeekAtLastError()`).
Attention : appel asynchrone donc `cudaDeviceSynchronize()` requis.
Attention 2 : `cudaDeviceSynchronize()` peut retourner une erreur également.

Programmation côté `kernel`

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 GPU car pas de pile.
Le code des fonctions est donc mis "inline" à la compilation.

Identification des threads/blocs

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

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
```

Qualificateurs des variables

mémoire	qualifier
registers	
shared	`__shared__`
local	`__local__`
global	`__device__`
constant	`__constant__`

Tableaux

Les tableaux sont stockés dans la mémoire locale, pas dans les registres.

Différents types de mémoire

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

Mémoire shared

- accès aussi rapide que des registres sous certaines conditions.
- quantité limité (16kio à 48kio).
- structurée en banques (16 banques pour Fermi).

Contraintes pour les performances

- Des threads distincts doivent accèder 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.

Mémoire shared

2 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.
```

Synchronisation des threads

__syncthreads()

Synchronisation de tous les threads d'un bloc avec une unique primitive : `__syncthreads()`

Synchronisation des threads

Exemple

```
__global__ void kernel( int * t, std::size_t size )
{
  int tid = ...
  t[ tid ] += t[ ( tid + 1 ) % blockDim.x ];
  // probleme entre warps d'un meme bloc (>1 warp)
  // acces concurrents en lecture et ecriture
}

__global__ void kernel( int * t, std::size_t size )
{
  int tid = ...
  int tmp = t[ ( tid + 1 ) % blockDim.x ];
  __syncthreads();
  t[ id ] += tmp;
}
```

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

Events

Mesure des temps sur GPU.

```
cudaEvent_t start, stop;

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

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

cudaEventSynchronize( stop ); // attente fin evenement

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

Pour conclure...

Programmation simple...

- CUDA = C/C++ + quelques mots-clés
- compilation simple

mais optimisation difficile !

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

Avant de porter/développer un code sur GPU

Questions à se poser :

- Structures des données adaptées ?
- Schémas d'accès aux données à transformer ?
- Calculs SPMD ?
- Divergence des exécutions ?
- Investissement en temps ?

Optimisations

voir exemple de Mark Harris, *Optimizing Parallel Reduction in CUDA*