NVIDIA CUDA

Compute Unified Device Architecture

Sylvain Jubertie

Laboratoire d'Informatique Fondamentale d'Orléans

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

NVIDIA CUDA

Compute Unified Device Architecture

1. cartes graphiques Nvidia
2. + pilotes CUDA
3. + extension langage C/C++
4. + compilateur

Installation : Logiciels

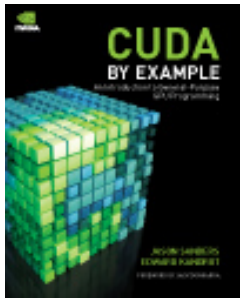
Téléchargement sur le site `www.nvidia.com/object/cuda_get.html`

1. Pilote Nvidia
2. CUDA Toolkit : compilateur, debugger, documentation
3. Nvidia GPU computing SDK : exemples CUDA + OpenCL
4. Nvidia NSight (Visual Studio ou Eclipse)
5. 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) : processeur ARM multicoeurs + GPU Nvidia

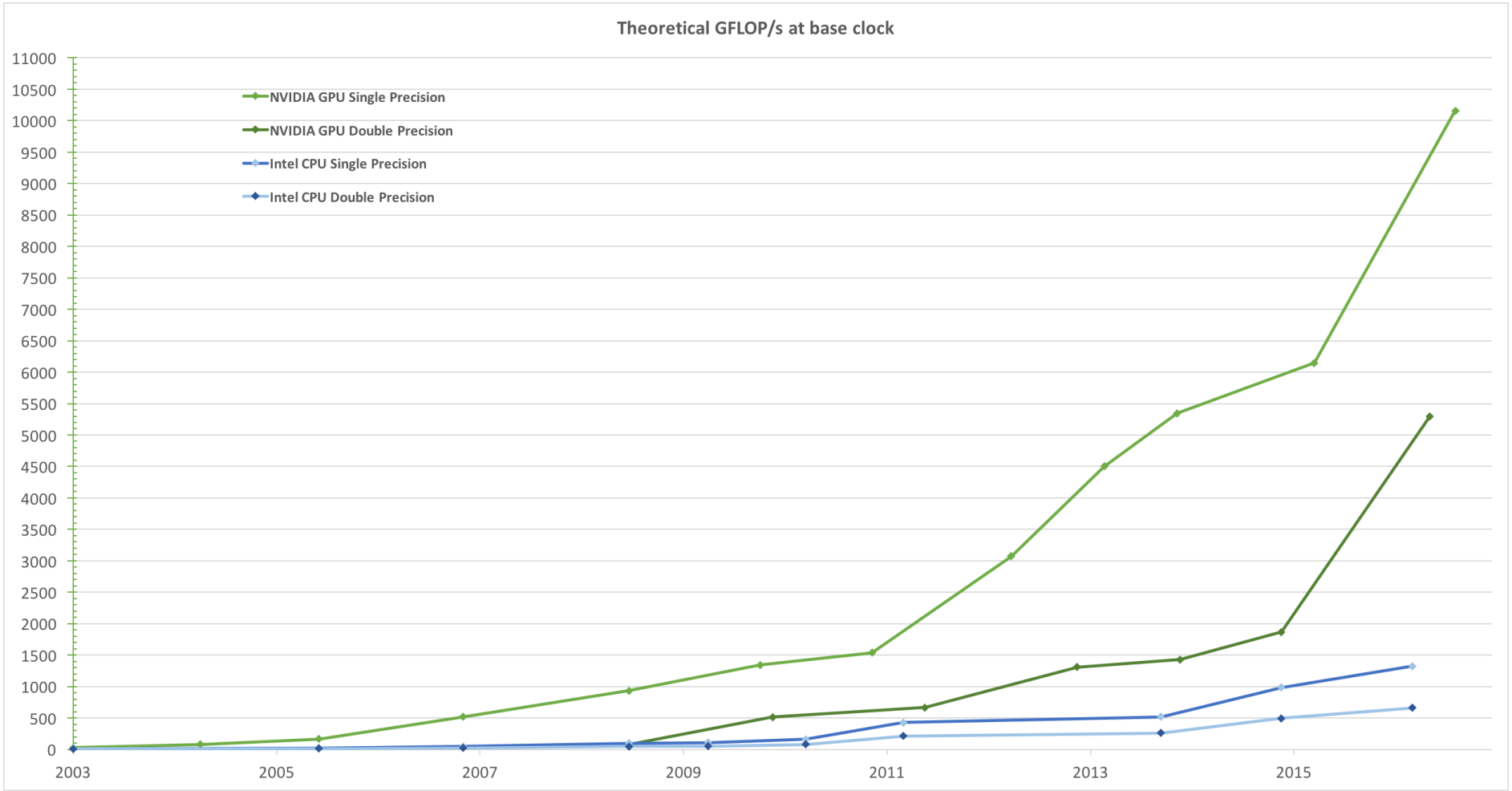
Gamme Tesla

Pourquoi utiliser des GPU ?

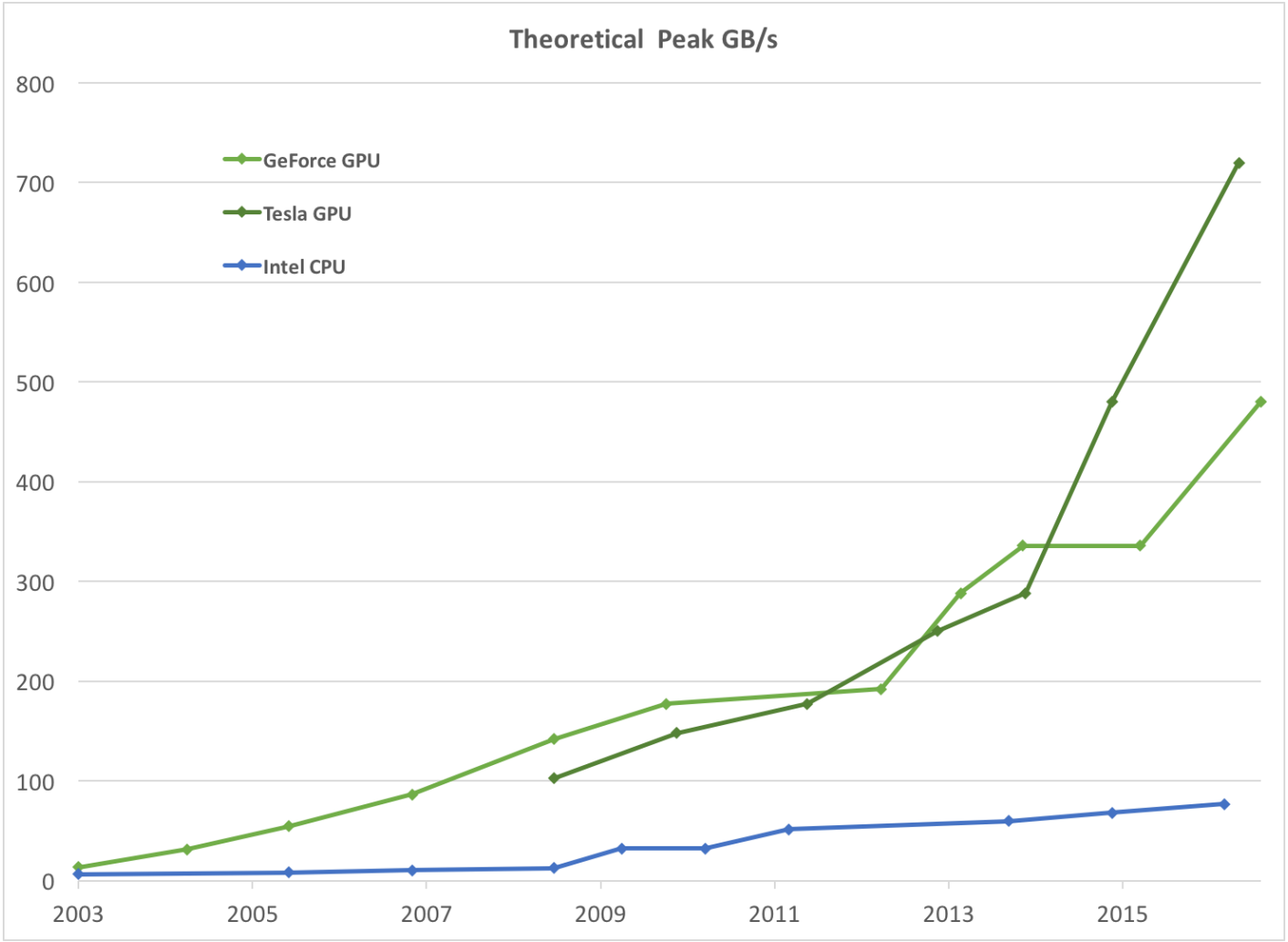
1. performances : architecture many-core
2. rapport performances / énergie
3. rapport performances / prix

	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.

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 ?

1 Introduction

2 Architecture

3 Modèle de programmation

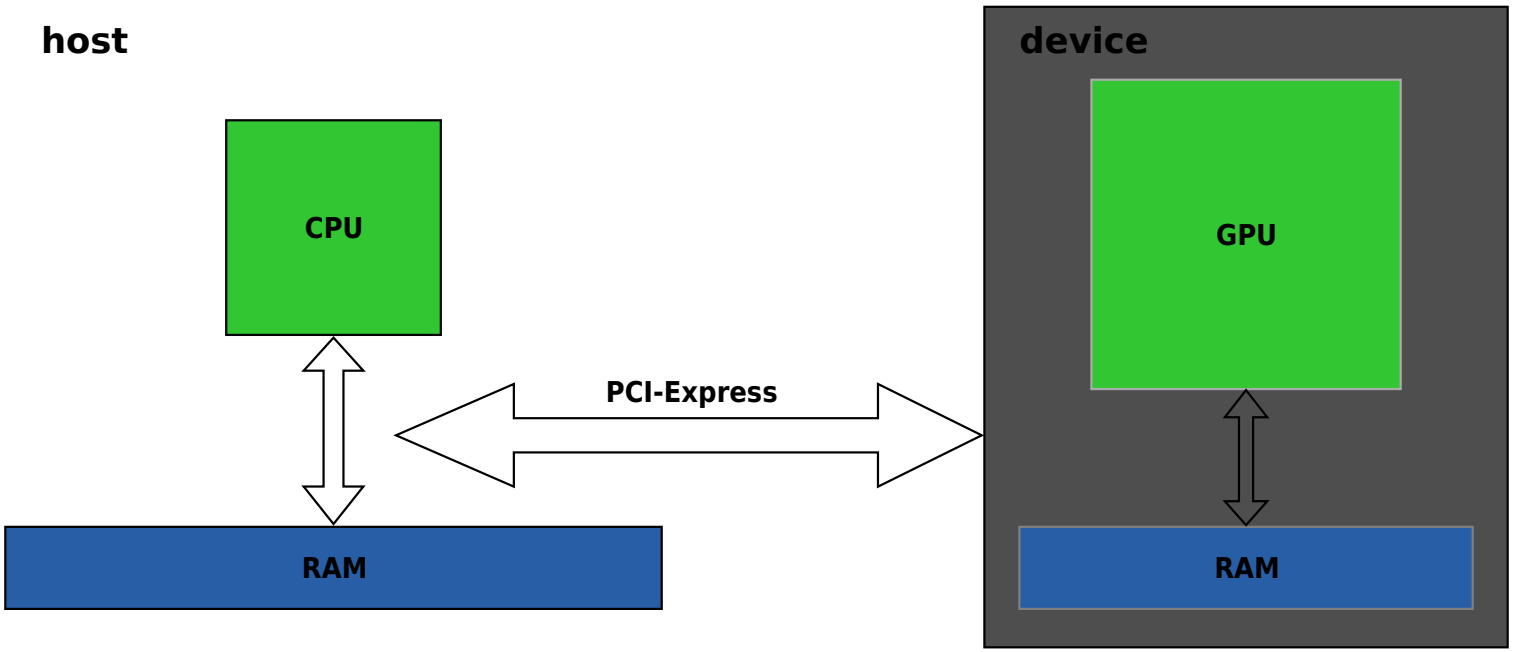
4 Modèle d'exécution

5 Programmation

6 Optimisation

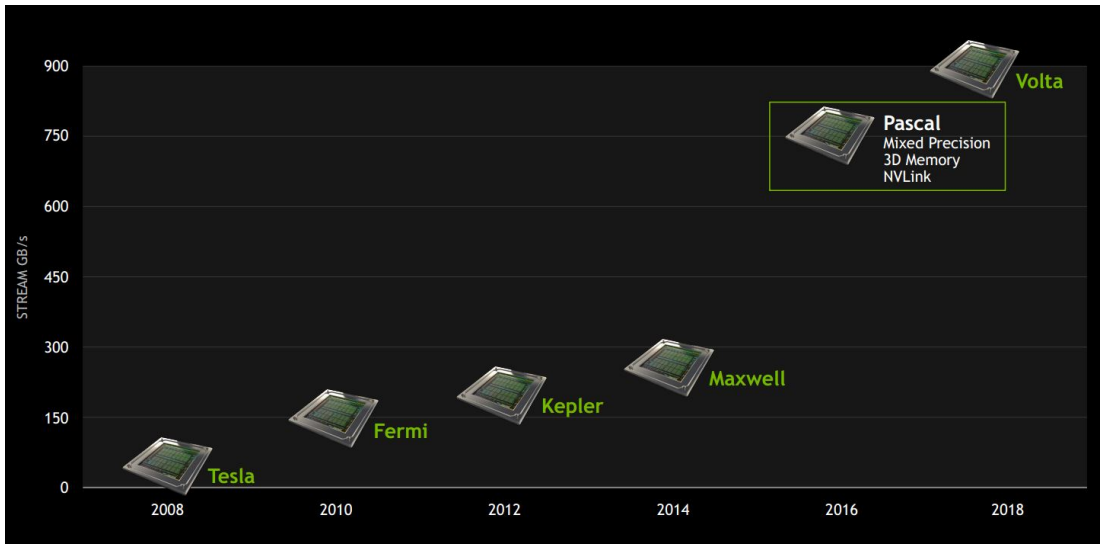
7 Bilan / Perpectives

Architecture générale



Générations de processeurs GPU

- Tesla (2008)
- Fermi (2010)
- Kepler (2012)
- Maxwell (2014)
- Pascal (2016)



Variantes

Chaque génération de processeurs possède des variantes :

- Tesla : G80, G84, G86, G92a/b, G98, GT200a/b
- Fermi : GF100, GF104, GF106, GF108
- Kepler : GK104, GK110 (À venir)
- 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 processeur
   - registres
   - mémoire partagée
   - caches : constantes et textures
2. dans la DRAM
   - mémoire 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

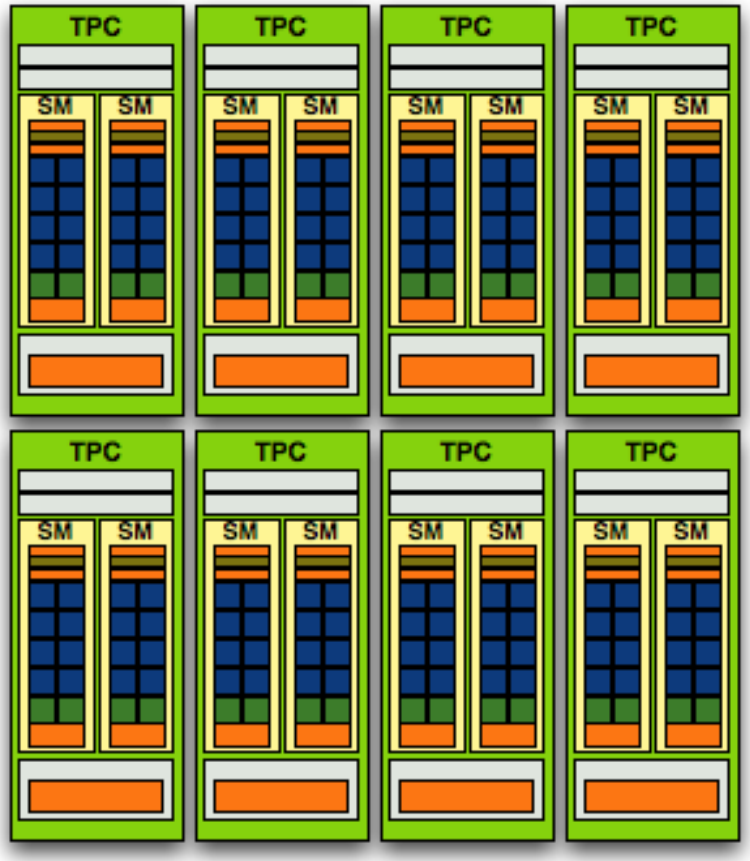
Architecture Tesla

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)**.

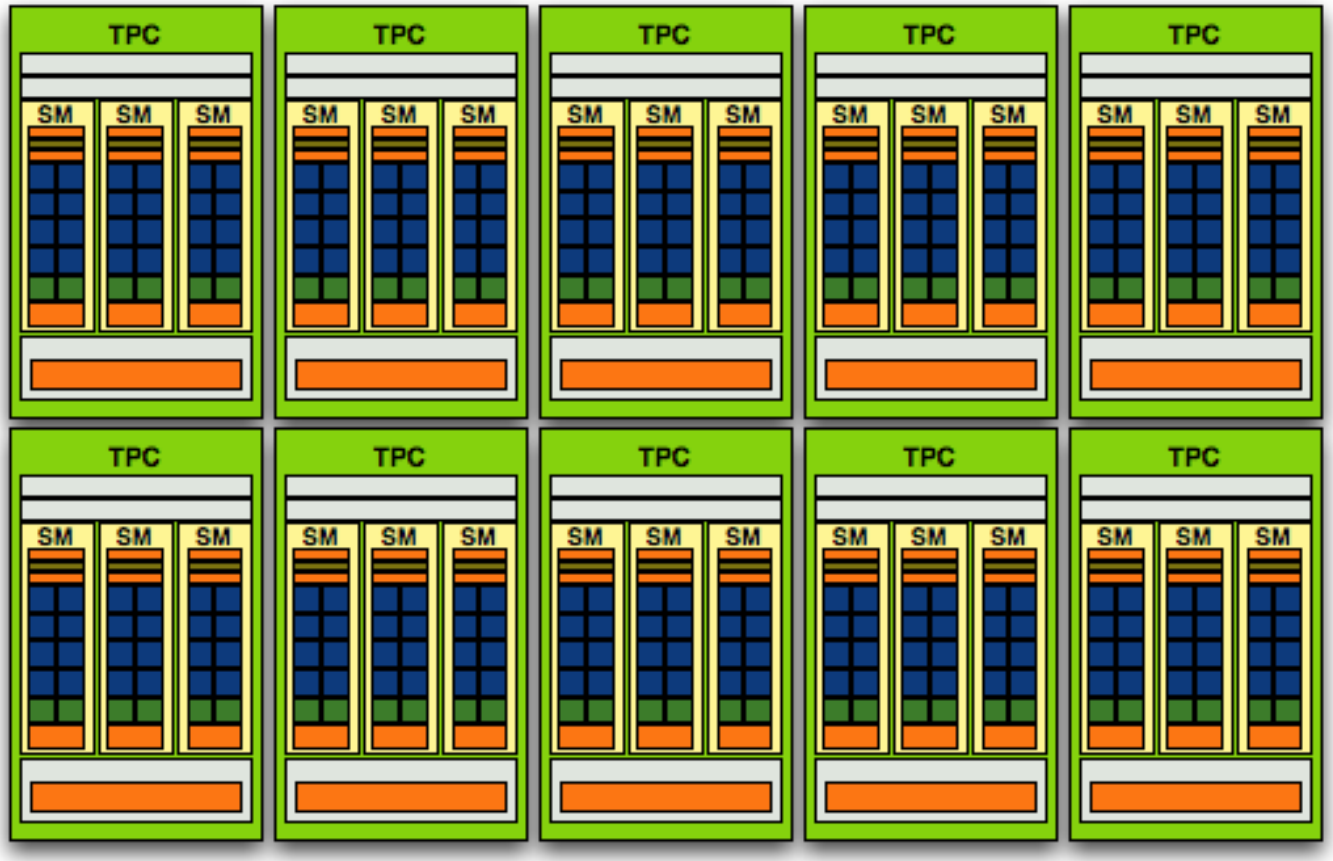
Exemple : Geforce GTX260

216 processeurs décomposés en 27 multi-processeurs.

Architecture Tesla (G80)



Architecture Tesla (GT200)



Architecture Tesla - Mémoires

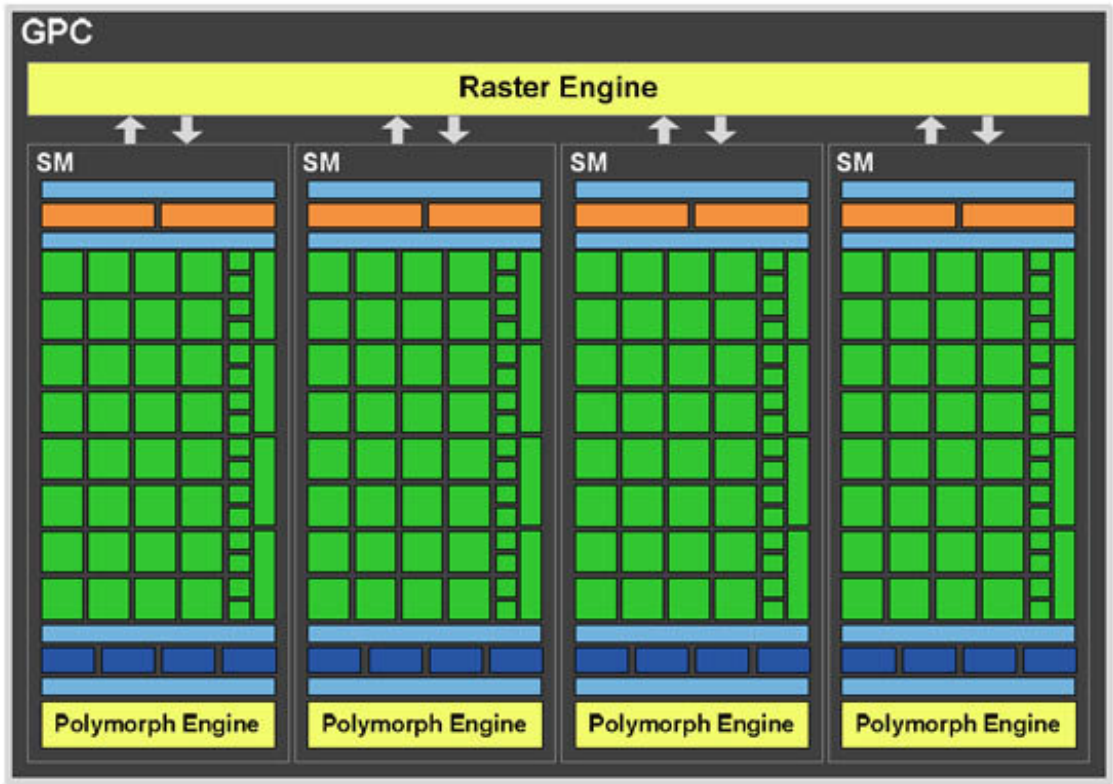
1. registres
2. mémoire partagée : par multi-processeur 16 KB divisés en 16 banques
3. mémoire globale
4. mémoire constante : 64 KB

Architecture Fermi

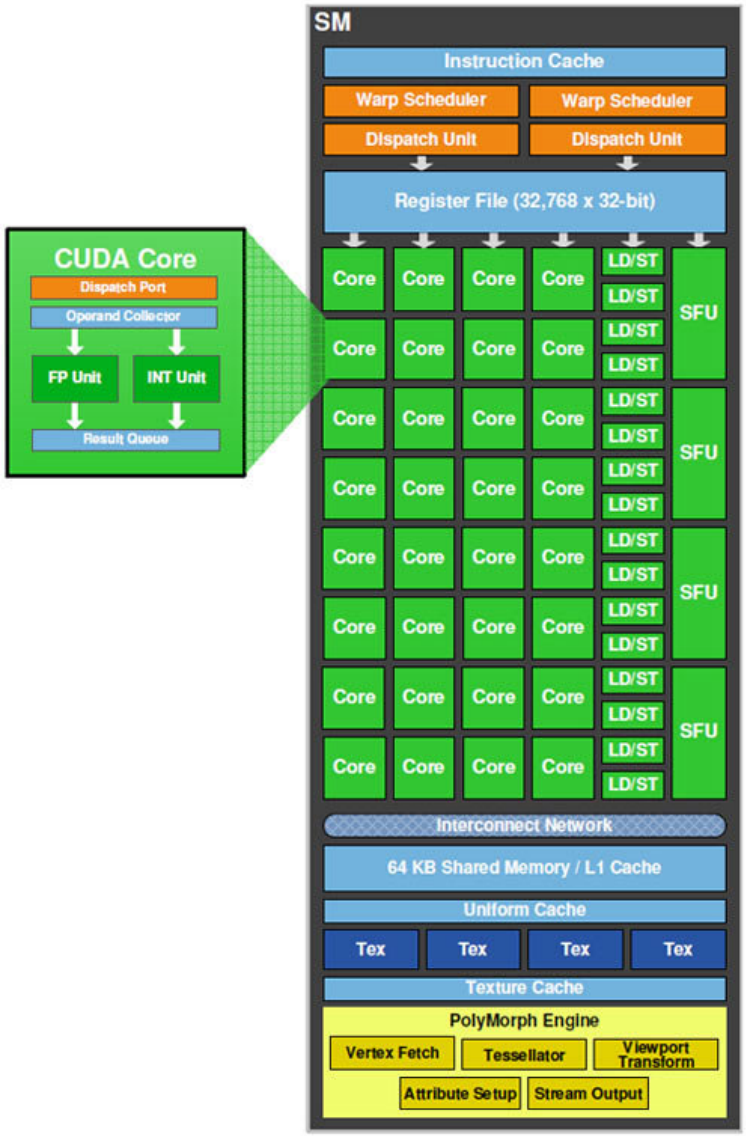
Les processeurs sont regroupés par 32 pour former des multi-processeurs.
Un GPU doté de 512 processeurs est donc composé de 16 multi-processeurs.
Les multi-processeurs sont regroupés par 4 au sein de **Graphics Processing Clusters**.

Architecture Fermi

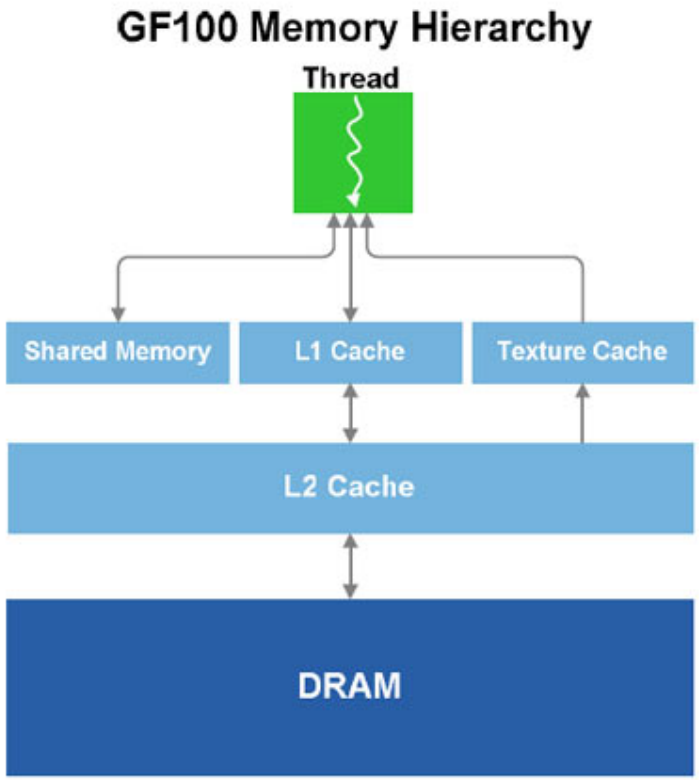
Architecture Fermi - GPC



Architecture Fermi - Multiprocesseur



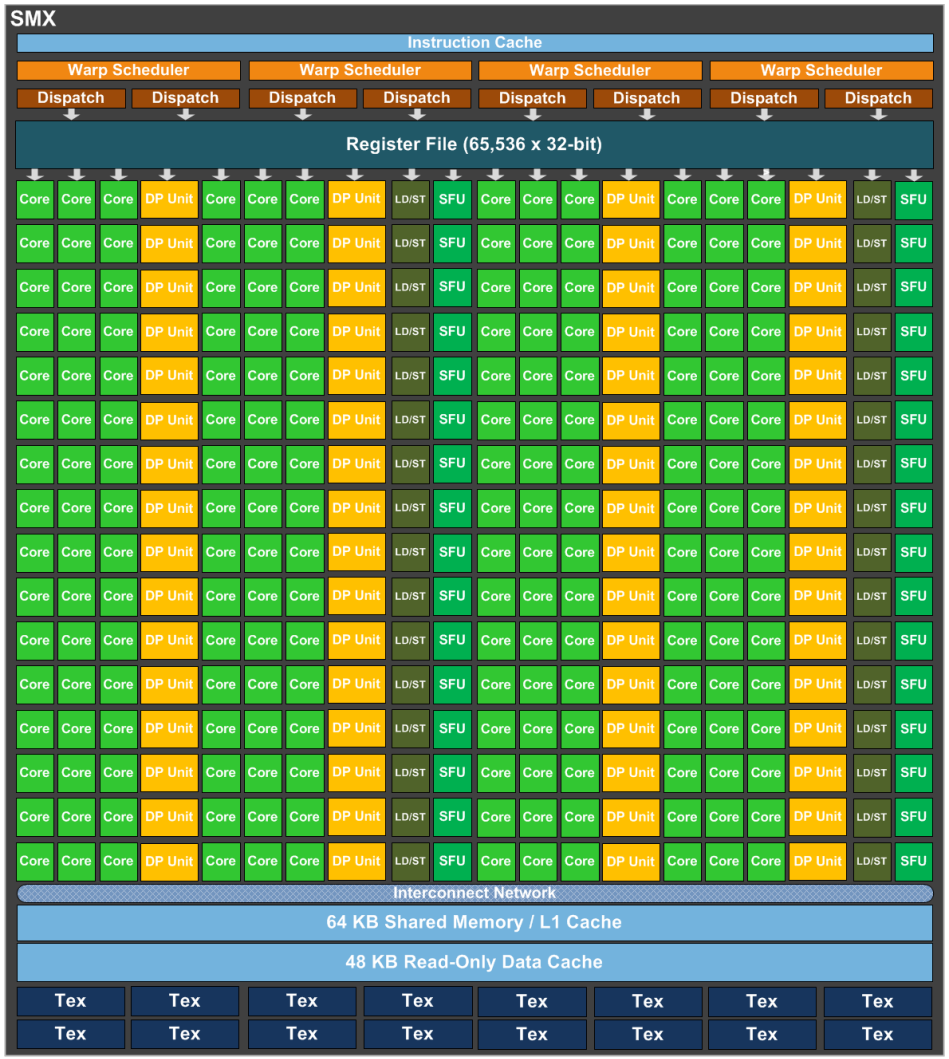
Architecture Fermi - Mémoires



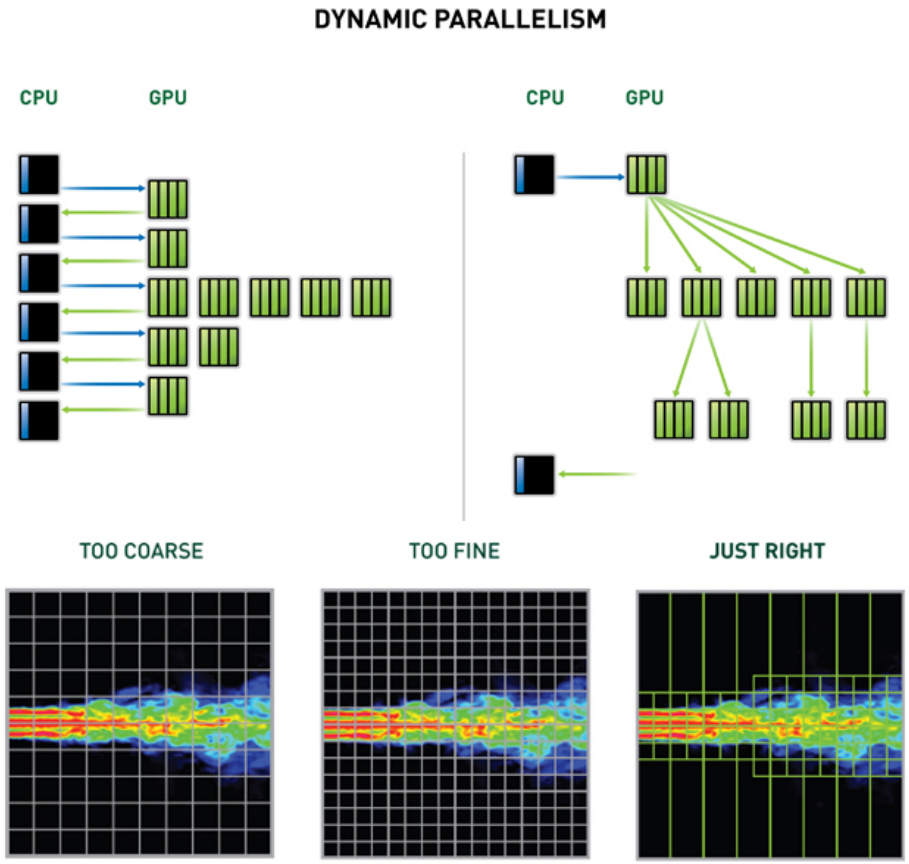
Architecture Kepler

- 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.
- ...

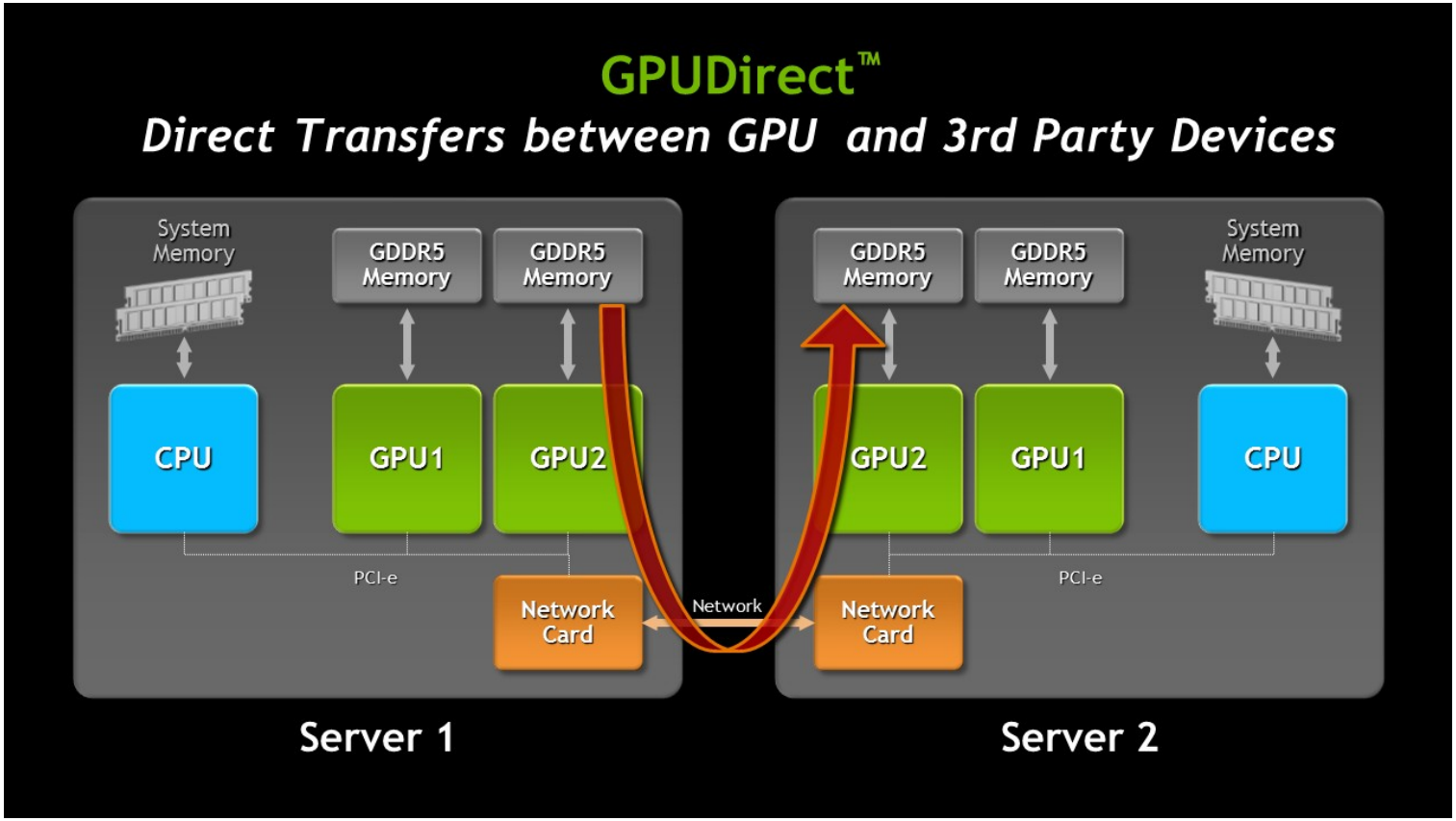
Architecture Kepler - SMX



Architecture Kepler - Parallélisme dynamique



Architecture Kepler - GPUDirect



Architecture Maxwell

...work in progress...

Architecture Pascal

Architectures : comparatif

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Pour l'instant...

- plusieurs générations d'architectures différentes : Tesla(discontinued), Fermi, Kepler et Maxwell
- codes CUDA (presque) portables d'une architecture à l'autre mais...
- optimisations spécifiques à chaque architecture !
- **Performances non portables !**
- nécessite une connaissance approfondie des architectures.
- mêmes problèmes que pour OpenCL.

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

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.

Le code **host** contrôle l'exécution du code **device** ainsi que les communications entre la mémoire **host** et **device**.
Un **kernel** est une procédure (pas de valeur de retour) destinée à être exécutée par le GPU.

```
__global__ void kernel( ... )
{
  // device code
}

int main()
{
  // host code
  ...
  // appel au kernel
  kernel<<< ... >>>( ... );
  ...
}
```

Code **host**

Déroulement typique

1. initialisation des mémoires
   - initialisation des données en mémoire **host**
   - allocation de 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**
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`

Code **device** : **kernel**

Data parallélisme

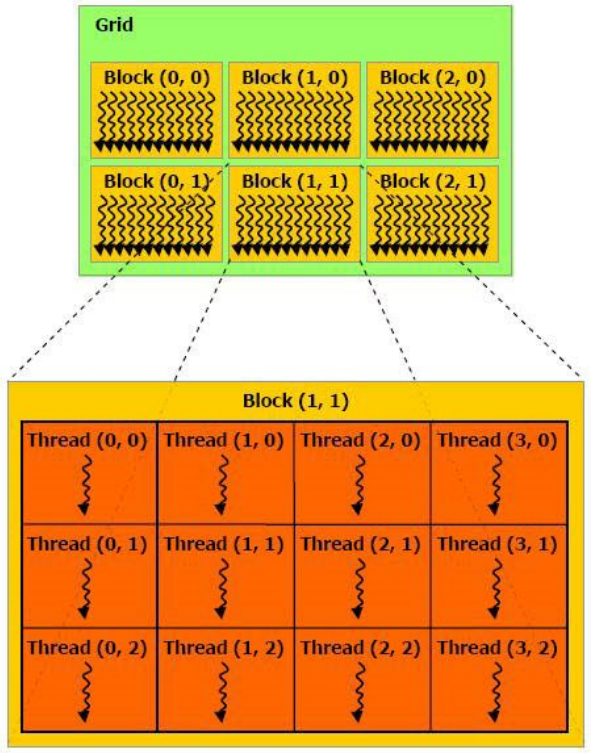
Modèle SPMD : Single Program on Multiple Data.

Thread

Une instance de kernel est appelée **thread**.
Chaque **thread** possède des identifiants propres 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.

Organisation des threads

Les threads peuvent être regroupés en blocs, eux-mêmes regroupés en grilles, chacun possédant un identifiant propre, éventuellement en 2 ou 3 dimensions, on parle alors des coordonnées d'un thread ou d'un bloc.



```
__global__ void kernel( ... )
{
   // Identifiant du thread
   unsigned int id = threadIdx.x;
}

int main()
{
   ...
   // Lancement de 16 threads ( 1 bloc de 16 threads )
   kernel<<< 1, 16 >>>( ... );
   ...
}
```

```
__global__ void kernel( ... )
{
  // Identifiant global 2D du thread
  unsigned int id = threadIdx.y * blockDim.x + threadI
}

int main()
{
  ...
  // Lancement de 256 threads ( 1 bloc de 16x16 thread
  dim3 threads( 16, 16 );
  kernel<<< 1, threads >>>( ... );
  ...
}
```

```
__global__ void kernel( ... )
{
   // Identifiant global du thread
   unsigned int id = blockIdx.x * blockDim.x + threadId
}

int main()
{
   ...
   // Lancement de 32 threads ( 2 bloc de 16 threads )
   kernel<<< 2, 16 >>>( ... );
   ...
}
```

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

Threads et processeurs

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

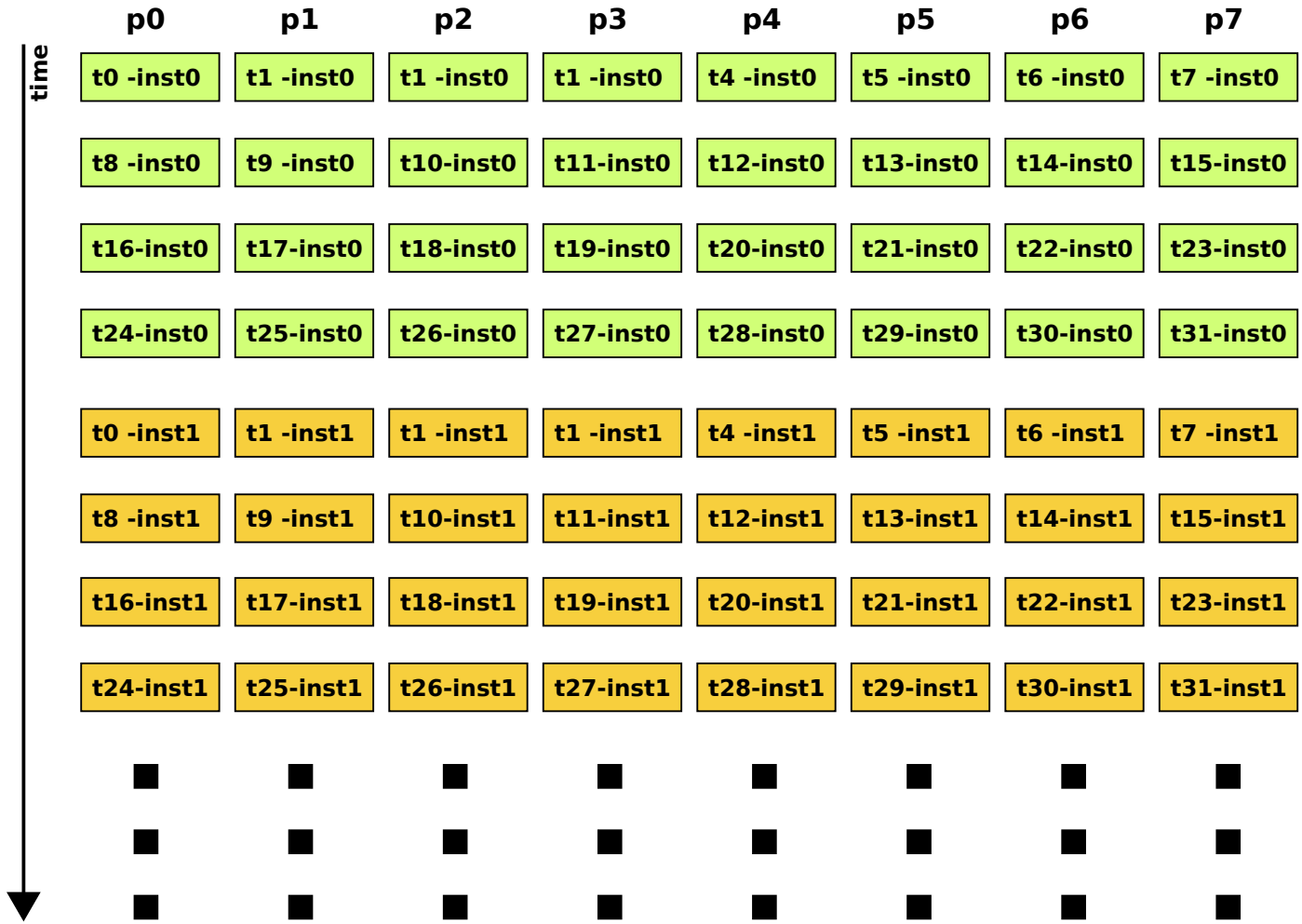
Les threads d'un même warp sont exécutés instruction par instruction : sur le Tesla, le multiprocesseur (8 processeurs sur Tesla) exécute la première instruction du kernel sur les 8 premiers threads simultanément puis passe au 8 suivants ...
Une fois l'instruction exécutée sur les 32 threads, on recommence 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.
Sur l'architecture **Fermi**, des blocs de kernels différents peuvent s'exécuter simultanément.

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

Premiers pas...

Exemple

Définition d'un kernel et appel depuis le code host :

```
__global__ void kernel() {}

int main() {

   kernel<<<1, 1>>>();

   return 0;

}
```

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

Allocation de mémoire sur le **device**

`cudaMalloc(&ptr, size)`

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

`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 :

```
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

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.

Synchronisation des threads

- fonction __syncthreads
- fonctions atomiques

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 ?

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

Optimisation

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

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 Bilan / Perpectives

Bilan / Perspectives

- "Généralisation" des architectures GPU pour augmenter les performances : designs cartes graphiques et cartes destinées au calcul tendent à se différencier.
- Signe que le marché du calcul gagne en importance ?
- Intel tente de rentrer sur le marché avec une architecture manycore x86.
- Le nombre de coeurs des CPU stagne (8-16 coeurs sur les CPU haut de gamme).
- Mais toutes les architectures actuelles sont multicore, même processeurs embarqués (ARM).
- Problème de complexité des architectures/modèles de programmation (flagrant sur le Cell IBM).
- L'évolution des GPU nécessite de réécrire le code pour l'optimiser (plus important que sur CPU).
- Mais le travail d'optimisation peut être transposé sur CPU.
- Limitations logiciels (API, compilateurs, ...).

Questions ?