NVIDIA CUDA

Compute Unified Device Architecture

Sylvain Jubertie

Laboratoire d'Informatique Fondamentale d'Orléans

2011-2012

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 CUDA avancé

8 Exemples

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 CUDA avancé

8 Exemples

NVIDIA CUDA

Compute Unified Device Architecture

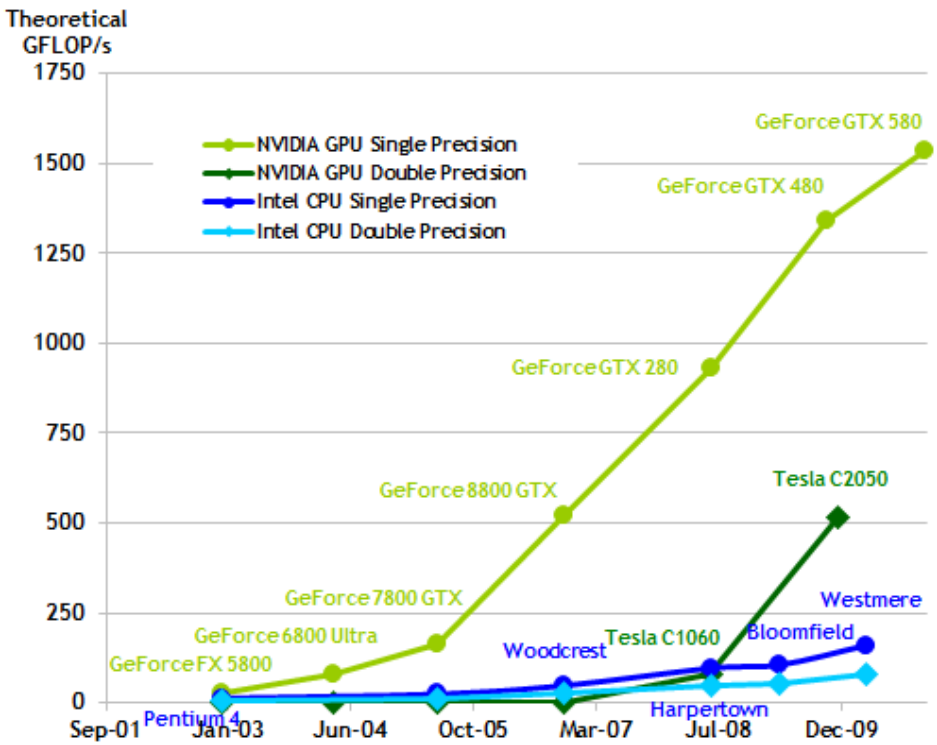
1. cartes graphiques Nvidia
2. + pilotes CUDA
3. + extension langage C/C++
4. + compilateur

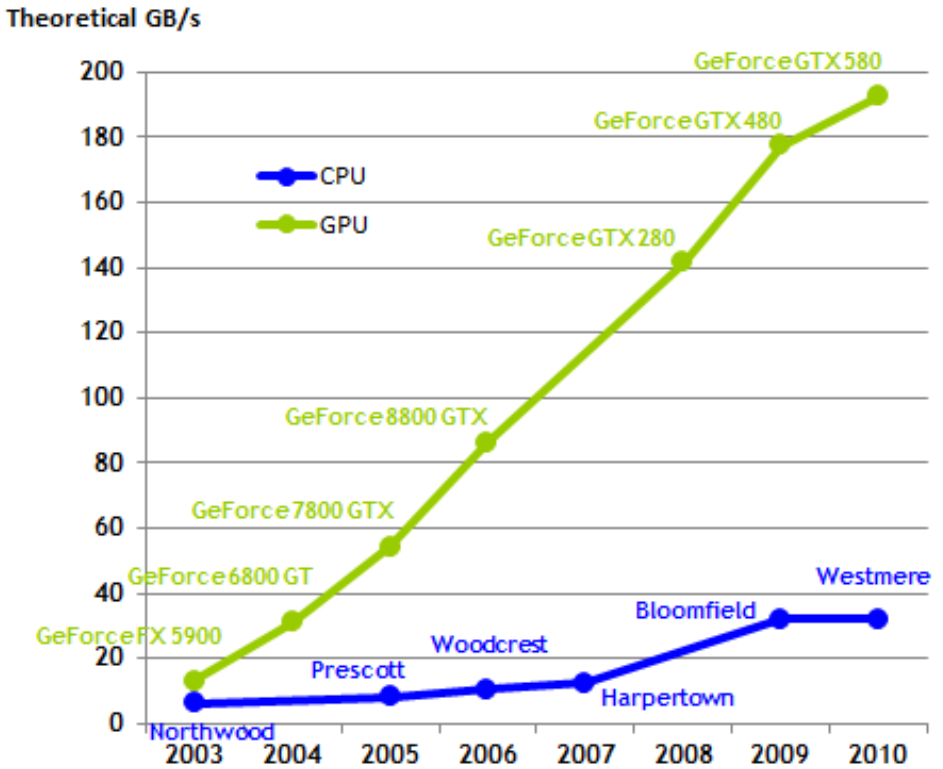
Gammes Nvidia

- GeForce : gamme grand public
- Quadro : gamme professionnelle pour la 3D : 3D stéréo (quad-buffer).
- Tesla : Gamme professionnelle pour le GPGPU : pas/moins de composants vidéo.
- ION & ION2 : gamme pour netbooks : jusqu'à 16 coeurs.
- Tegra 4 ? : gamme pour les plateformes nomades (tablettes, smartphones) : processeur ARM multicoeurs + GPU Nvidia

Pourquoi utiliser des GPU ?

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

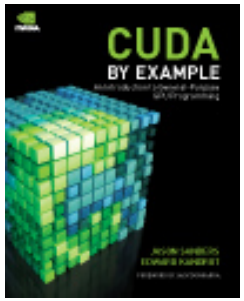




Ressources : Livres

- 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

Installation : Logiciels

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

1. Pilote Nvidia
2. CUDA Toolkit : compilateur, debugger
3. Nvidia GPU computing SDK : exemples CUDA + OpenCL

1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

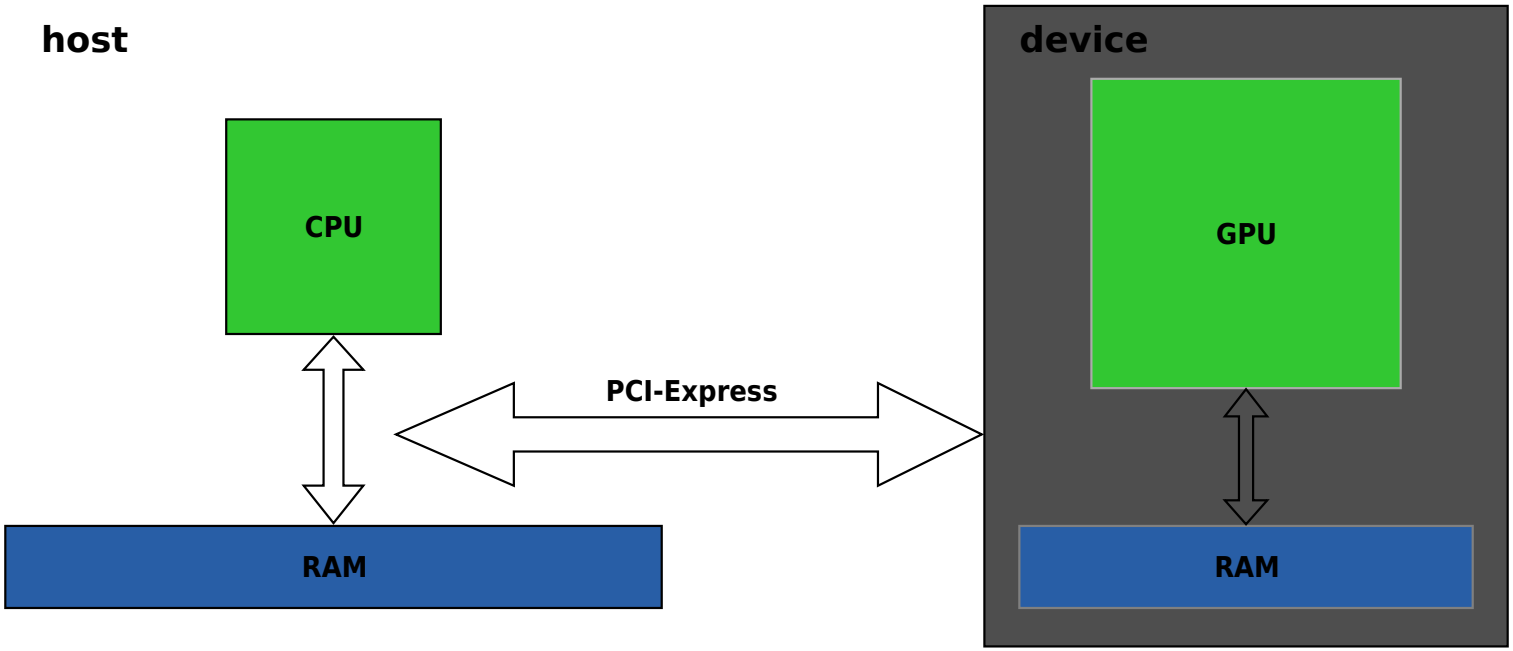
5 Programmation

6 Optimisation

7 CUDA avancé

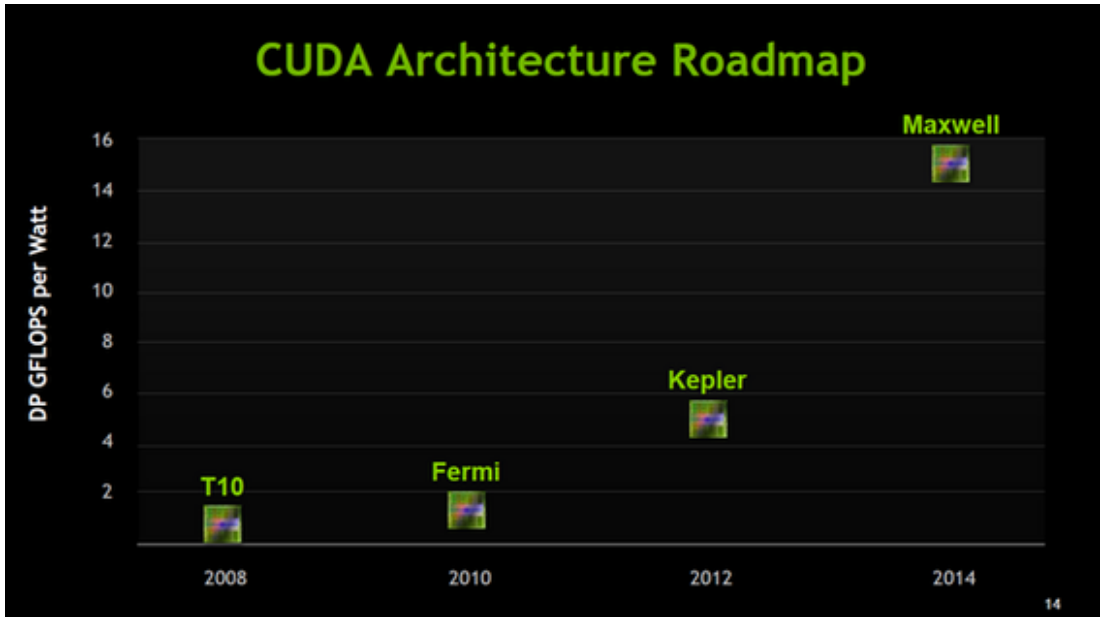
8 Exemples

Architecture générale



Générations de processeurs GPU

- Tesla (2008)
- Fermi (2010)
- À venir : Kepler (2012) & Maxwell (2014)



Variantes

Chaque génération de processeurs possède des variantes :

- Tesla : G80, G84, G86, G92a/b, G98, GT200a/b
- Fermi : GF100, GF104, GF106, GF108

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*.

Compute Capabilities

Modèle	Compute capability
Geforce GTX580	2.0
Geforce GTX560	2.1
Geforce GT260	1.3
Geforce GT240	1.2
Geforce 9800GT	1.1
Geforce 8800GTS	1.0

Pour l'instant...

- 2 générations d'architectures différentes : Tesla et Fermi
- code CUDA portable d'une architecture à l'autre mais...
- optimisations spécifiques à chaque architecture !
- **Performances non portables !**
- nécessite une connaissance approfondie des architectures.

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

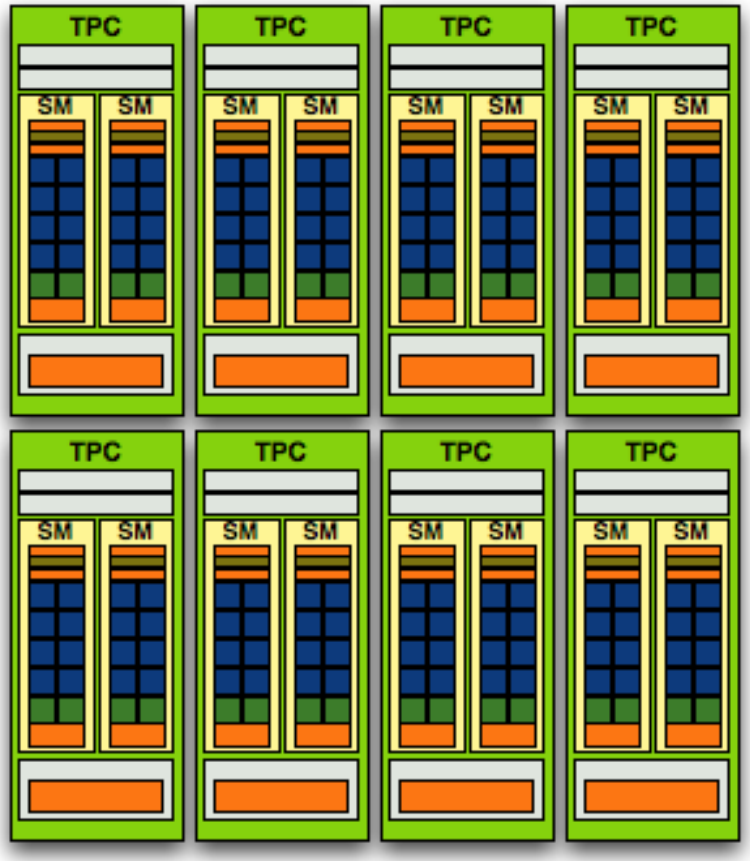
Architecture Tesla

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)**.

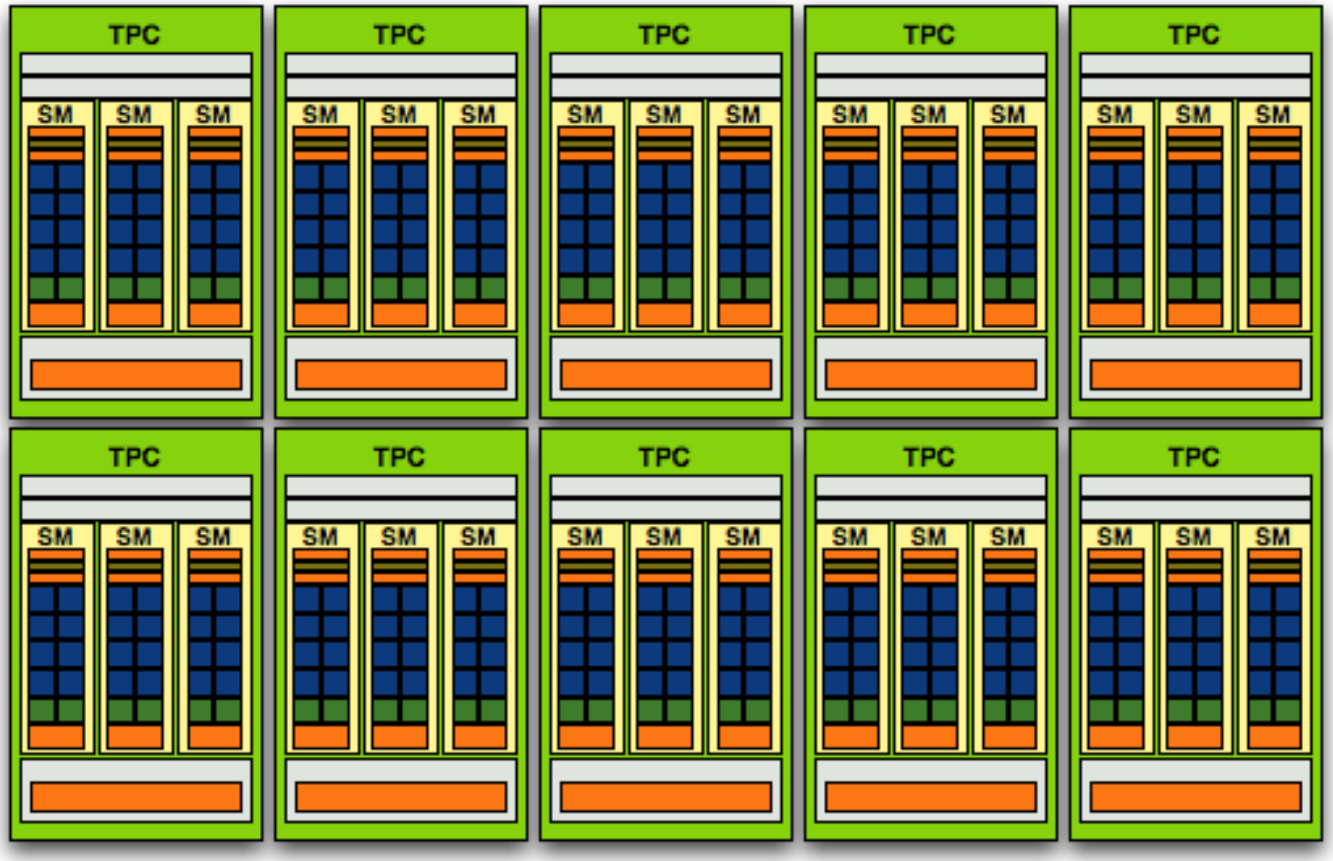
Exemple : Geforce GTX260

216 processeurs décomposés en 27 multi-processeurs.

Architecture Tesla (G80)



Architecture Tesla (GT200)

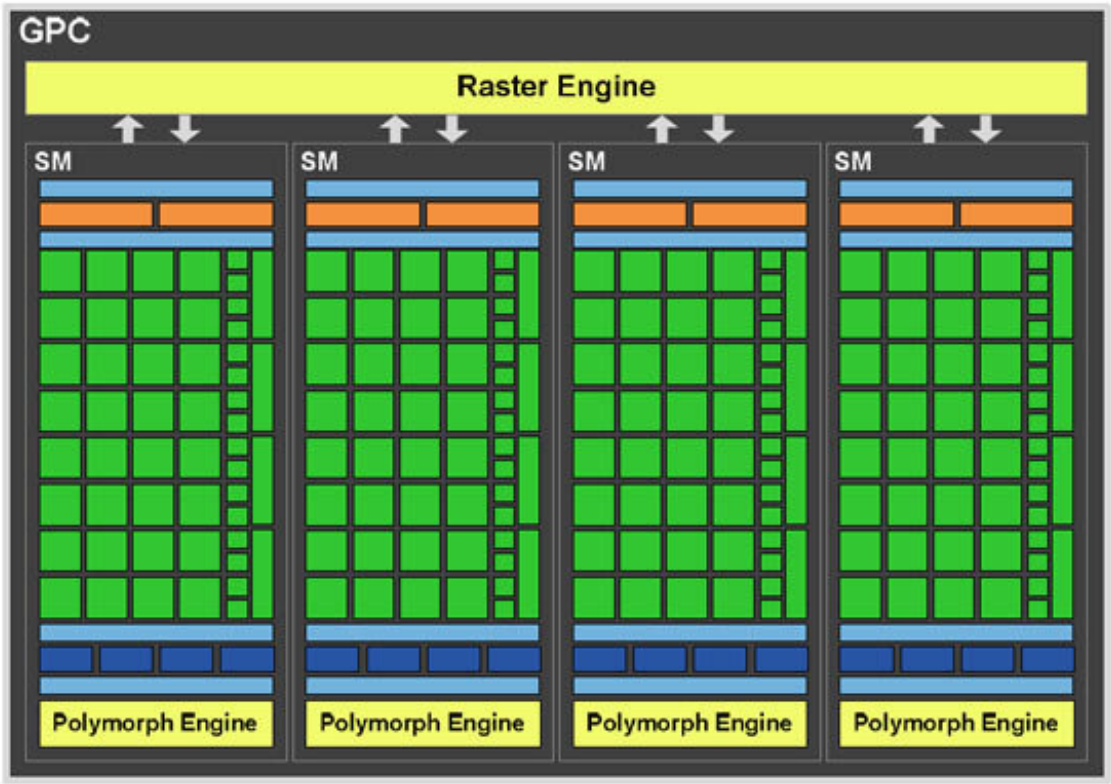


Architecture Fermi

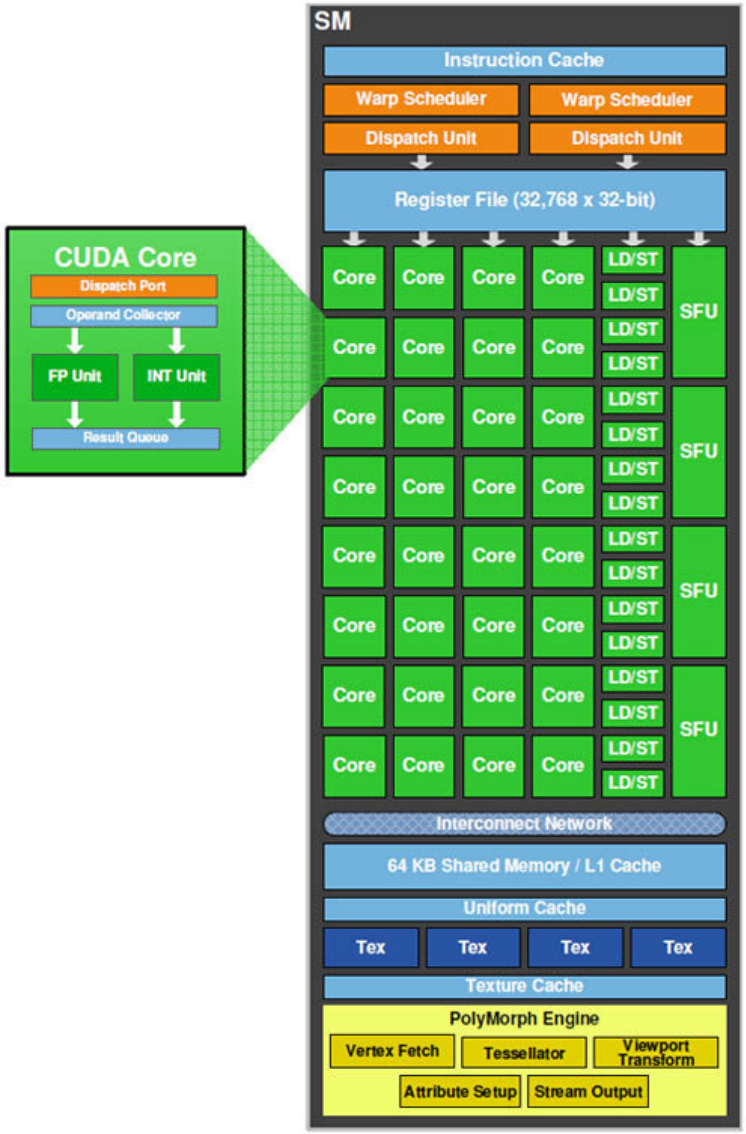
Les processeurs sont regroupés par 32 pour former des multi-processeurs.
Un GPU doté de 512 processeurs est donc composé de 16 multi-processeurs.
Les multi-processeurs sont regroupés par 4 au sein de **Graphics Processing Clusters**.

Architecture Fermi

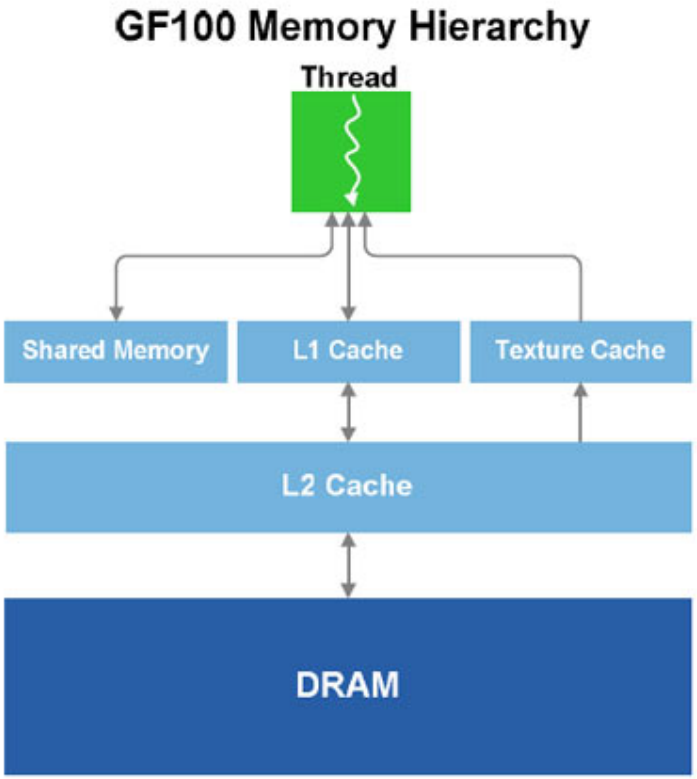
Architecture Fermi - GPC



Architecture Fermi - Multiprocesseur



Architecture Fermi - Mémoires



1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 CUDA avancé

8 Exemples

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.

Code **host**

Déroulement typique

1. initialisation des mémoires
   - initialisation des données en mémoire **host**
   - allocation de la mémoire globale **device**
2. copie des données de la mémoire host vers la mémoire **device**
3. exécution du **kernel** sur les données en mémoire **device**
4.
   - copie des résultats en mémoire **device** vers la mémoire **host**
   - exploitation directe des résultats par OpenGL pour affichage
5. libération de la mémoire globale **device**

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

Data parallélisme

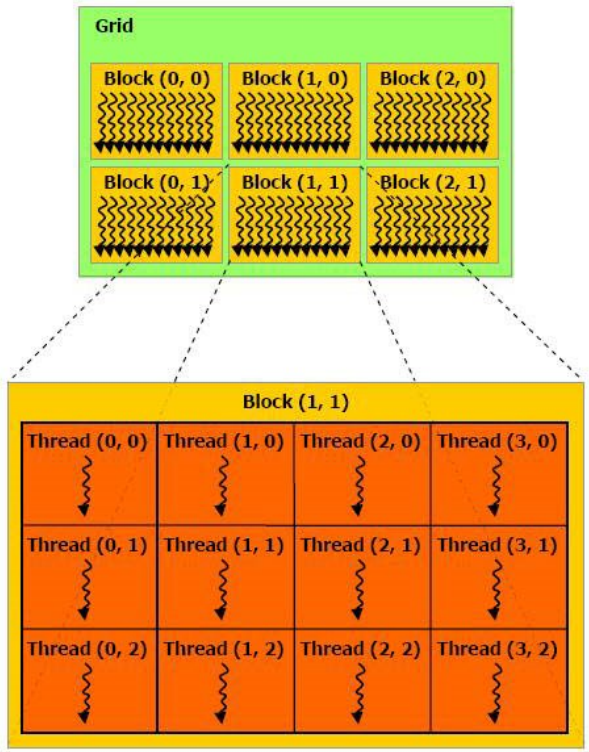
Modèle SPMD : Single Program on Multiple Data.

Thread

Une instance de kernel est appelée **thread**.
Chaque **thread** possède un identifiant propre 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.



1 Introduction

2 Architecture

3 Modèle de programmation

4 Modèle d'exécution

5 Programmation

6 Optimisation

7 CUDA avancé

8 Exemples

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 ou 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

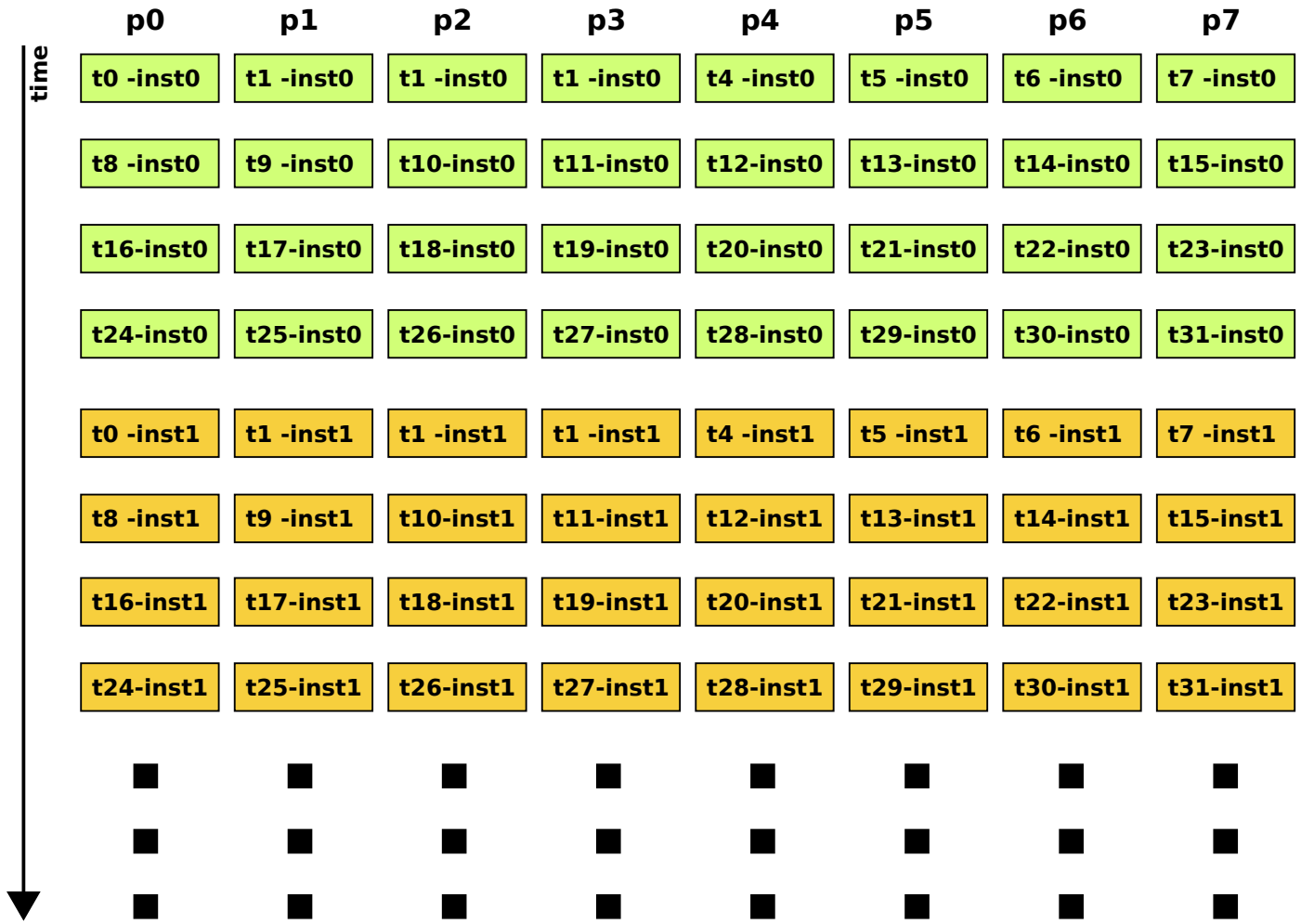
Les threads d'un même warp sont exécutés instruction par instruction : sur le Tesla, le multiprocesseur (8 processeurs sur Tesla) exécute la première instruction du kernel sur les 8 premiers threads simultanément puis passe au 8 suivants ...
Une fois l'instruction exécutée sur les 32 threads, on recommence 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 CUDA avancé

8 Exemples

Premiers pas...

Exemple

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

```
__global__ void kernel() {}

int main() {

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

   return 0;

}
```

Code device

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.

Code 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

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

Différents types de mémoire

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

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