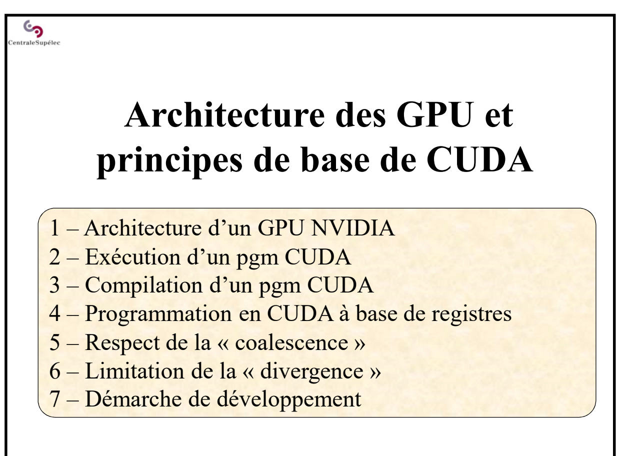
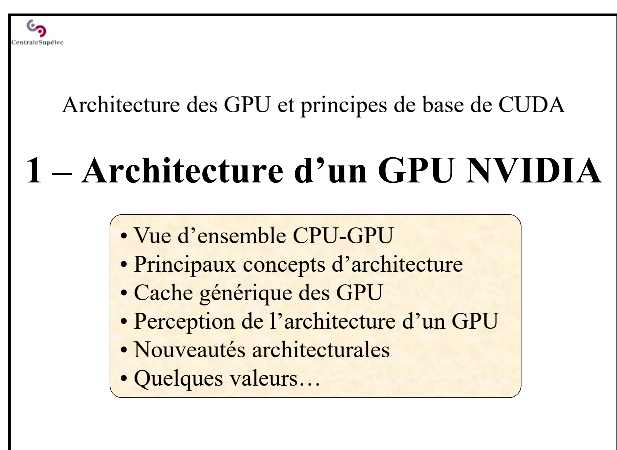
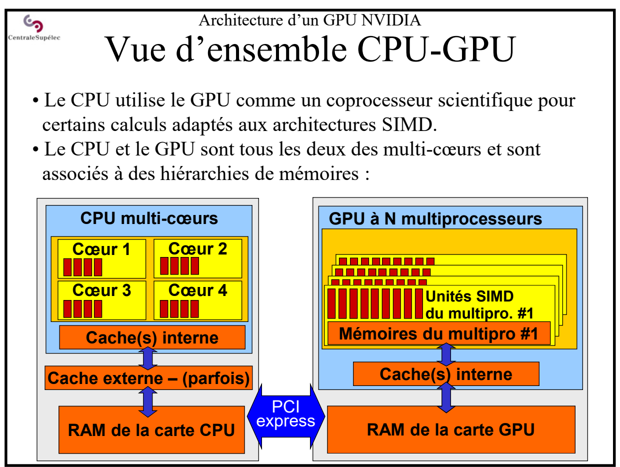
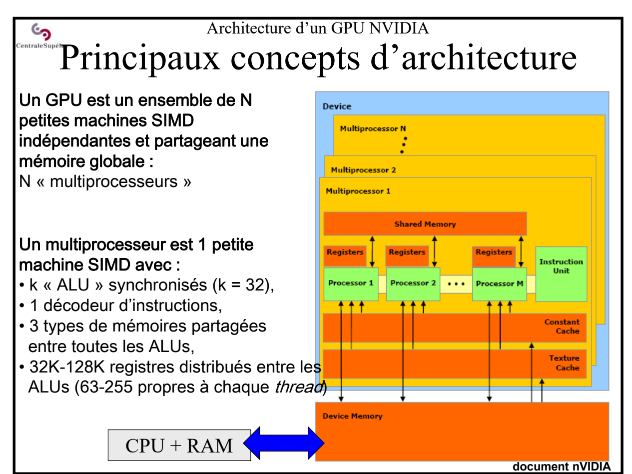
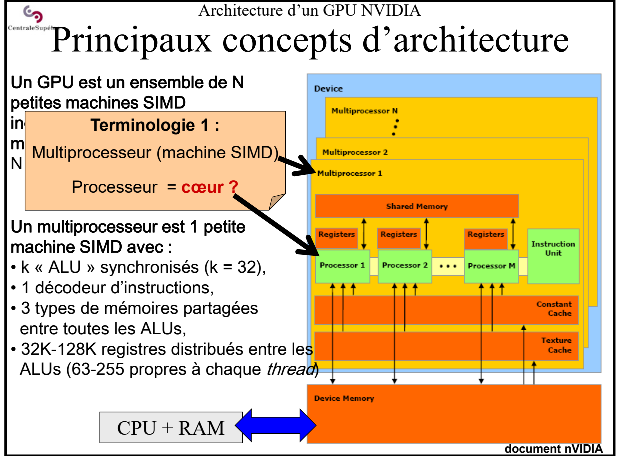
CentraleSupélec
Mineure HPC-SBD
Architecture des GPU et principes de base de CUDA
Stéphane Vialle
université PARIS-SACLAY
Sciences et technologies de l'information et de la communication (STIC)
RISEGrid
Grand Est
Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

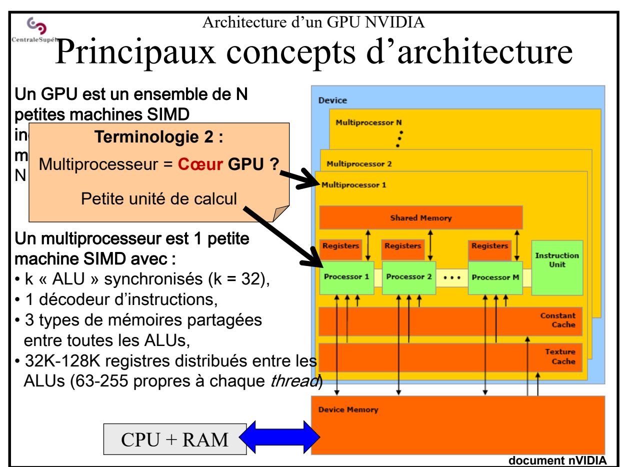

Architecture des GPU et principes de base de CUDA
1 – Architecture d'un GPU NVIDIA
2 – Exécution d'un pgm CUDA
3 – Compilation d'un pgm CUDA
4 – Programmation en CUDA à base de registres
5 – Respect de la « coalescence »
6 – Limitation de la « divergence »
7 – Démarche de développement

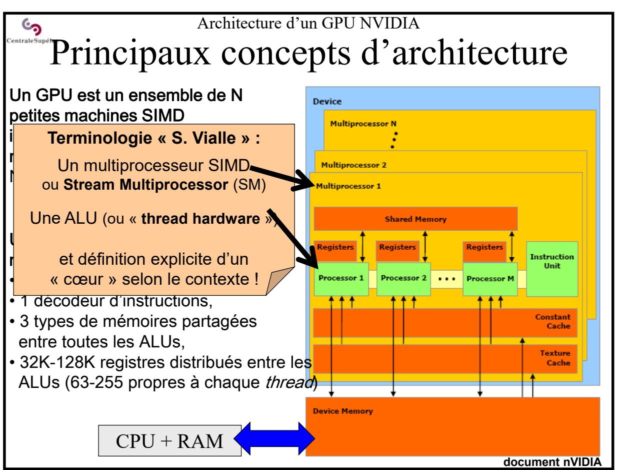

Architecture des GPU et principes de base de CUDA
1 – Architecture d'un GPU NVIDIA
• Vue d'ensemble CPU-GPU
• Principaux concepts d'architecture
• Cache générique des GPU
• Perception de l'architecture d'un GPU
• Nouveautés architecturales
• Quelques valeurs…

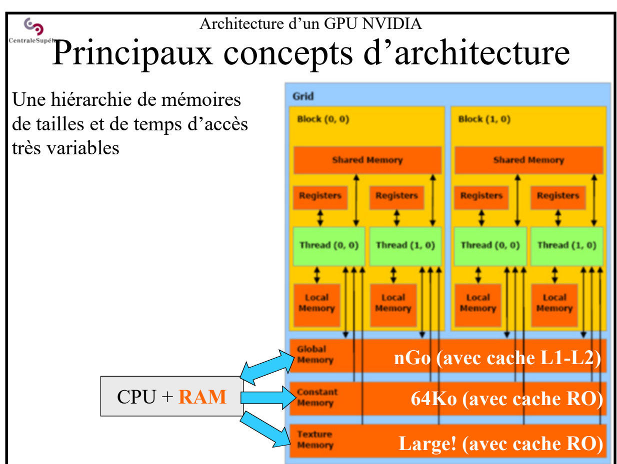

Architecture d'un GPU NVIDIA
Vue d'ensemble CPU-GPU
• Le CPU utilise le GPU comme un coprocesseur scientifique pour certains calculs adaptés aux architectures SIMD.
• Le CPU et le GPU sont tous les deux des multi-cœurs et sont associés à des hiérarchies de mémoires :
CPU multi-cœurs
Cœur 1
Cœur 2
Cœur 3
Cœur 4
Cache(s) interne
Cache externe – (parfois)
RAM de la carte CPU
PCI express
GPU à N multiprocesseurs
Unités SIMD du multipro. #1
Mémoires du multipro #1
Cache(s) interne
RAM de la carte GPU

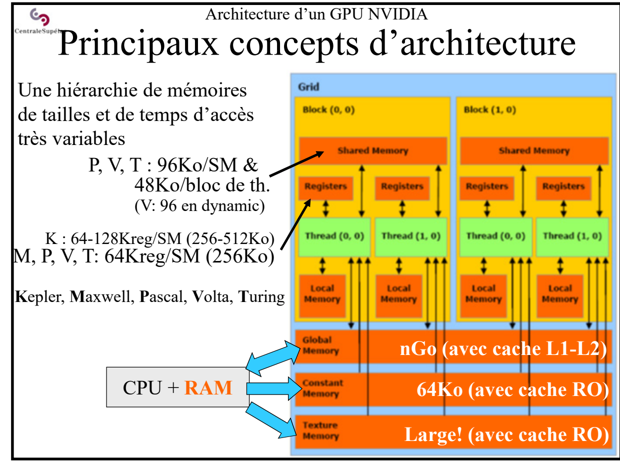

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Un GPU est un ensemble de N petites machines SIMD indépendantes et partageant une mémoire globale :
N « multiprocesseurs »
Un multiprocesseur est 1 petite machine SIMD avec :
• k « ALU » synchronisés (k = 32),
• 1 décodeur d'instructions,
• 3 types de mémoires partagées entre toutes les ALUs,
• 32K-128K registres distribués entre les ALUs (63-255 propres à chaque thread)
CPU + RAM
document nVIDIA

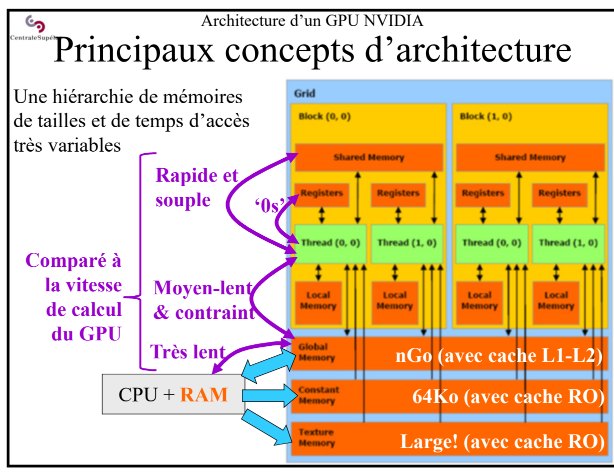

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Un GPU est un ensemble de N petites machines SIMD
Terminologie 1 :
Multiprocesseur (machine SIMD)
Processeur = cœur ?
Un multiprocesseur est 1 petite machine SIMD avec :
• k « ALU » synchronisés (k = 32),
• 1 décodeur d'instructions,
• 3 types de mémoires partagées entre toutes les ALUs,
• 32K-128K registres distribués entre les ALUs (63-255 propres à chaque thread)
CPU + RAM
document nVIDIA

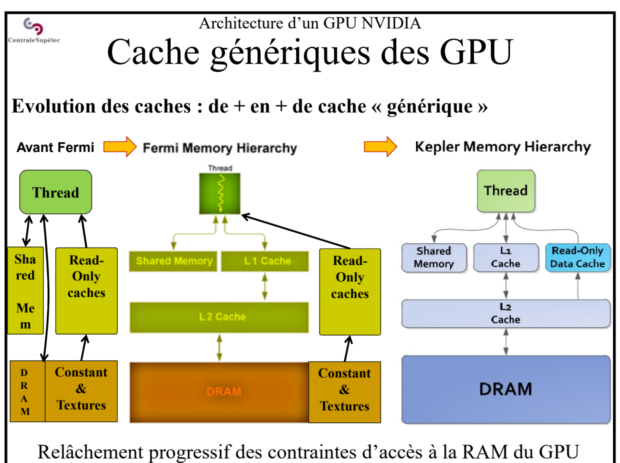

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Un GPU est un ensemble de N petites machines SIMD
Terminologie 2 :
Multiprocesseur = Cœur GPU ?
Petite unité de calcul
Un multiprocesseur est 1 petite machine SIMD avec :
• k « ALU » synchronisés (k = 32),
• 1 décodeur d'instructions,
• 3 types de mémoires partagées entre toutes les ALUs,
• 32K-128K registres distribués entre les ALUs (63-255 propres à chaque thread)
CPU + RAM
document nVIDIA

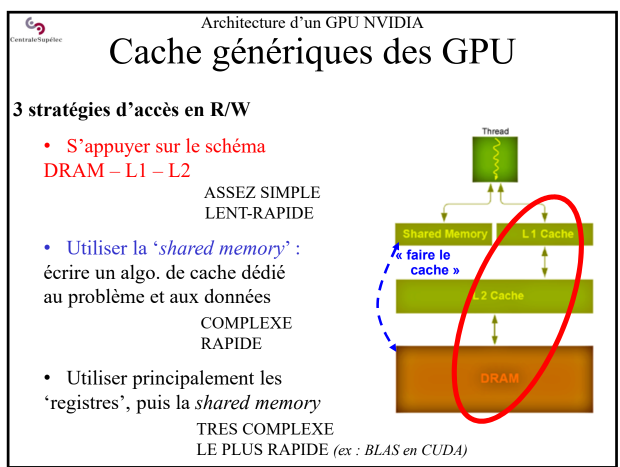

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Un GPU est un ensemble de N petites machines SIMD
Terminologie « S. Vialle » :
Un multiprocesseur SIMD ou Stream Multiprocessor (SM)
Une ALU (ou « thread hardware »)
et définition explicite d'un « cœur » selon le contexte !
• 1 décodeur d'instructions,
• 3 types de mémoires partagées entre toutes les ALUs,
• 32K-128K registres distribués entre les ALUs (63-255 propres à chaque thread)
CPU + RAM
document nVIDIA

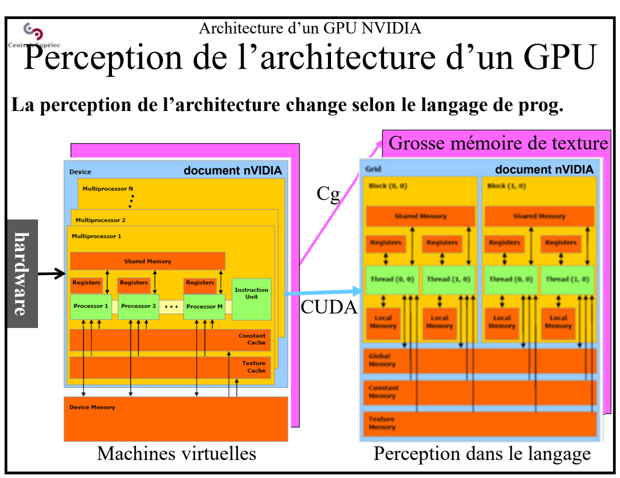

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Une hiérarchie de mémoires de tailles et de temps d'accès très variables
nGo (avec cache L1-L2)
64Ko (avec cache RO)
Large! (avec cache RO)
CPU + RAM

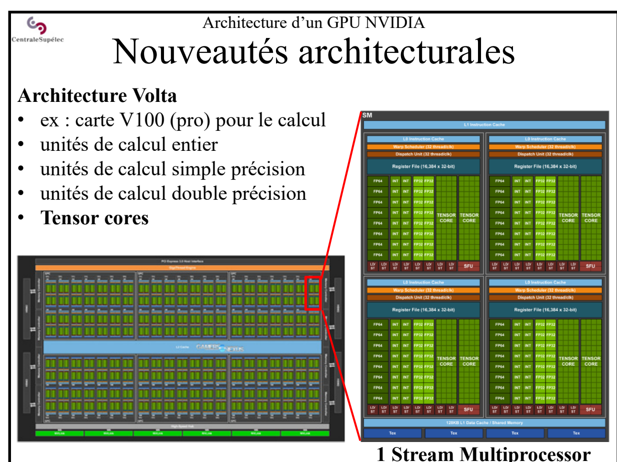

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Une hiérarchie de mémoires de tailles et de temps d'accès très variables
P, V, T : 96Ko/SM & 48Ko/bloc de th.
(V: 96 en dynamic)
K : 64-128Kreg/SM (256-512Ko)
M, P, V, T: 64Kreg/SM (256Ko)
Kepler, Maxwell, Pascal, Volta, Turing
nGo (avec cache L1-L2)
64Ko (avec cache RO)
Large! (avec cache RO)
CPU + RAM

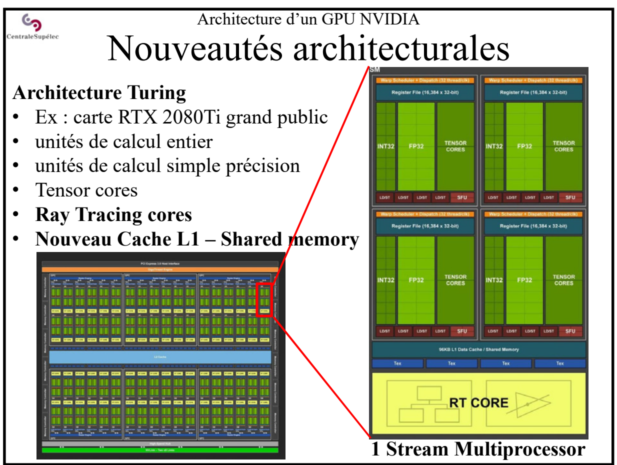

Architecture d'un GPU NVIDIA
Principaux concepts d'architecture
Une hiérarchie de mémoires de tailles et de temps d'accès très variables
Rapide et souple
'0s'
Comparé à la vitesse de calcul du GPU
Moyen-lent & contraint
Très lent
nGo (avec cache L1-L2)
64Ko (avec cache RO)
Large! (avec cache RO)
CPU + RAM

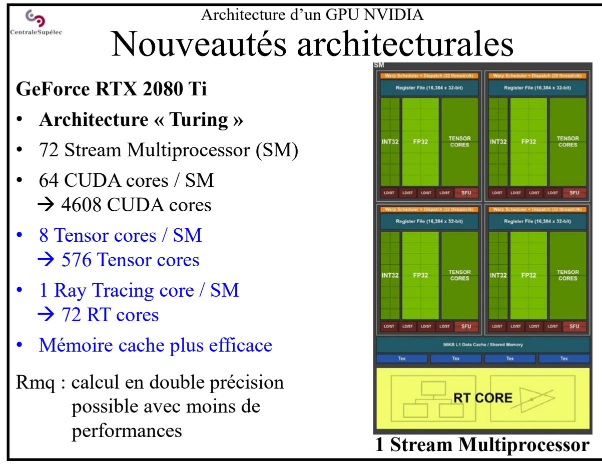

Architecture d'un GPU NVIDIA
Cache génériques des GPU
Evolution des caches : de + en + de cache « générique »
Avant Fermi
Fermi Memory Hierarchy
Kepler Memory Hierarchy
Thread
Shared Mem
Read-Only caches
DRAM
Constant & Textures
Shared Memory
L1 Cache
L2 Cache
DRAM
L1 Cache
Read-Only Data Cache
L2 Cache
Relâchement progressif des contraintes d'accès à la RAM du GPU

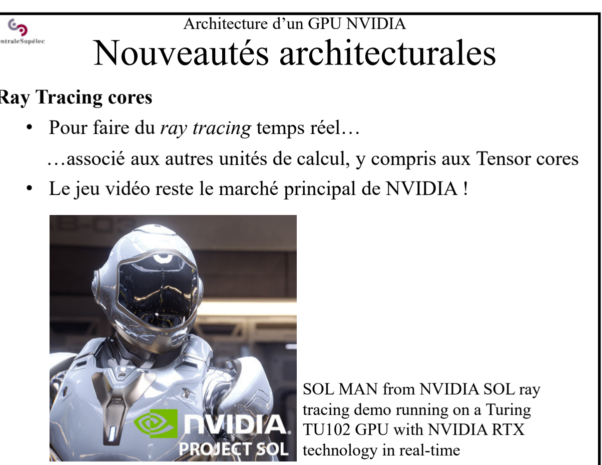

Architecture d'un GPU NVIDIA
Cache génériques des GPU
3 stratégies d'accès en R/W
S'appuyer sur le schéma DRAM – L1 – L2
ASSEZ SIMPLE
LENT-RAPIDE
Utiliser la 'shared memory' : écrire un algo. de cache dédié au problème et aux données
COMPLEXE
RAPIDE
Utiliser principalement les 'registres', puis la shared memory
TRES COMPLEXE
LE PLUS RAPIDE (ex : BLAS en CUDA)
Thread
Shared Memory
L1 Cache
« faire le cache »
L2 Cache
DRAM

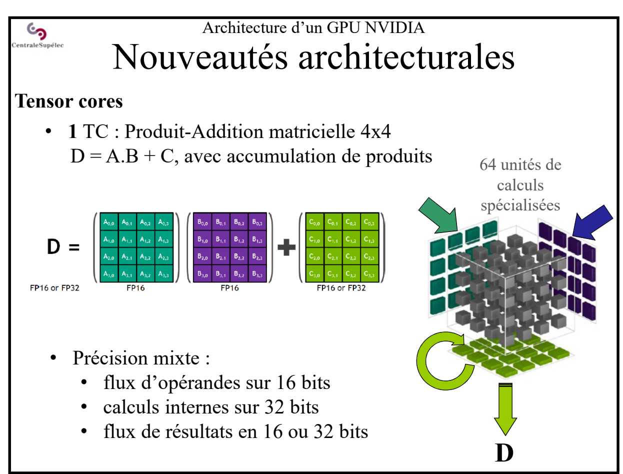

Architecture d'un GPU NVIDIA
Perception de l'architecture d'un GPU
La perception de l'architecture change selon le langage de prog.
Grosse mémoire de texture
hardware
document nVIDIA
document nVIDIA
Cg
CUDA
Machines virtuelles
Perception dans le langage

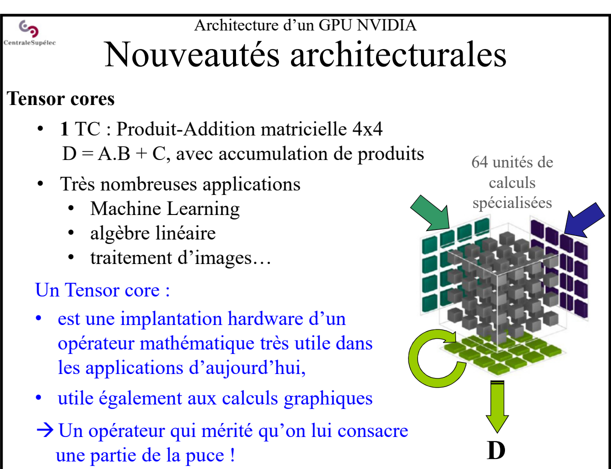

Architecture d'un GPU NVIDIA
Nouveautés architecturales
Architecture Volta
ex : carte V100 (pro) pour le calcul
unités de calcul entier
unités de calcul simple précision
unités de calcul double précision
Tensor cores
1 Stream Multiprocessor

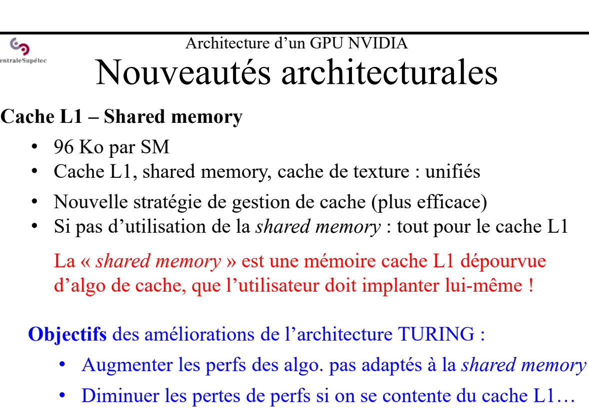

Architecture d'un GPU NVIDIA
Nouveautés architecturales
Architecture Turing
Ex : carte RTX 2080Ti grand public
unités de calcul entier
unités de calcul simple précision
Tensor cores
Ray Tracing cores
Nouveau Cache L1 – Shared memory
RT CORE
1 Stream Multiprocessor

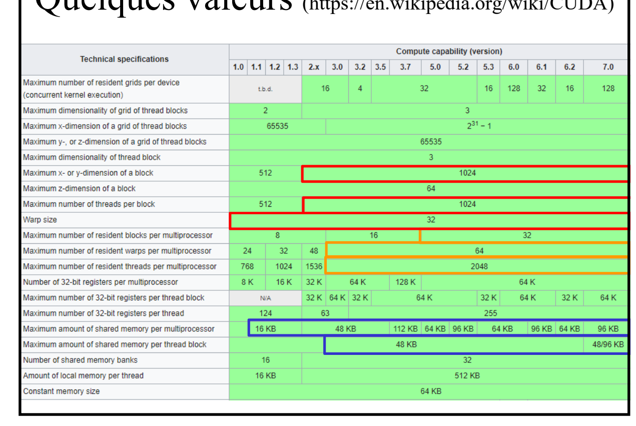

Architecture d'un GPU NVIDIA
Nouveautés architecturales
GeForce RTX 2080 Ti
Architecture « Turing »
72 Stream Multiprocessor (SM)
64 CUDA cores / SM
→ 4608 CUDA cores
8 Tensor cores / SM
→ 576 Tensor cores
1 Ray Tracing core / SM
→ 72 RT cores
Mémoire cache plus efficace
Rmq : calcul en double précision possible avec moins de performances
RT CORE
1 Stream Multiprocessor


Architecture d'un GPU NVIDIA
Nouveautés architecturales
Ray Tracing cores
Pour faire du ray tracing temps réel…
…associé aux autres unités de calcul, y compris aux Tensor cores
Le jeu vidéo reste le marché principal de NVIDIA !
NVIDIA
PROJECT SOL
SOL MAN from NVIDIA SOL ray tracing demo running on a Turing TU102 GPU with NVIDIA RTX technology in real-time



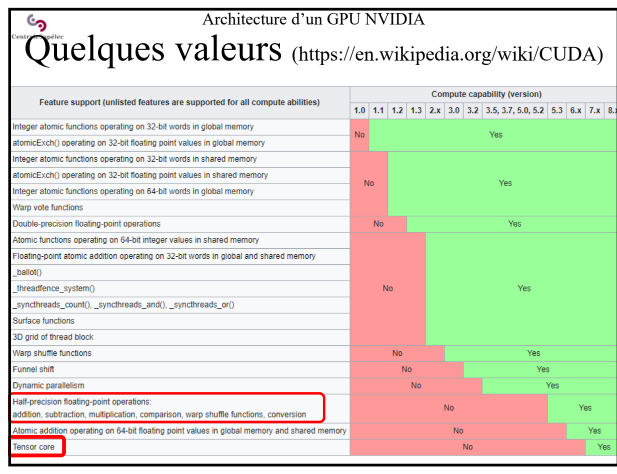





Architecture d'un GPU NVIDIA

Quelques valeurs (https://en.wikipedia.org/wiki/CUDA)

Technical specifications	Compute capability (version)															
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	2				3											
Maximum x-dimension of a grid of thread blocks	65535					$2^{31}-1$										
Maximum y-, or z-dimension of a grid of thread blocks	65535															
Maximum dimensionality of thread block	3															
Maximum x- or y-dimension of a block	512				1024											
Maximum z-dimension of a block	64															
Maximum number of threads per block	512				1024											
Warp size	32															
Maximum number of resident blocks per multiprocessor	8					16				32						
Maximum number of resident warps per multiprocessor	24		32		48	64										
Maximum number of resident threads per multiprocessor	768		1024		1536	2048										
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K						
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K	64 K				32 K	64 K		32 K	64 K
Maximum number of 32-bit registers per thread	124				63		255									
Maximum amount of shared memory per multiprocessor	16 KB				48 KB				112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB
Maximum amount of shared memory per thread block	48 KB															48/96 KB
Number of shared memory banks	16				32											
Amount of local memory per thread	16 KB				512 KB											
Constant memory size	64 KB															


Architecture d'un GPU NVIDIA

Quelques valeurs (https://en.wikipedia.org/wiki/CUDA)

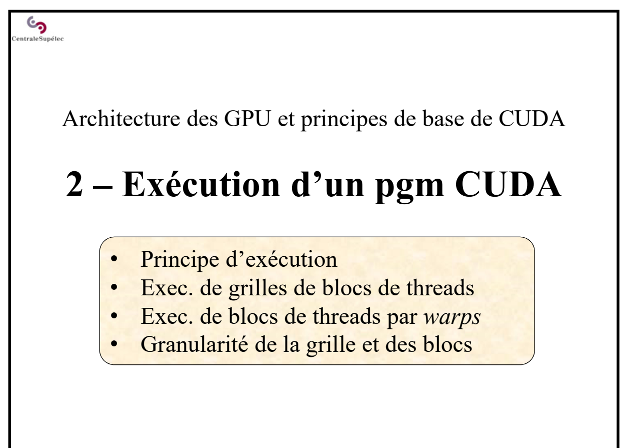
Feature support (unlisted features are supported for all compute abilities)	Compute capability (version)											
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.x
Integer atomic functions operating on 32-bit words in global memory	No	Yes										
atomicExch() operating on 32-bit floating point values in global memory												
Integer atomic functions operating on 32-bit words in shared memory	No		Yes									
atomicExch() operating on 32-bit floating point values in shared memory												
Integer atomic functions operating on 64-bit words in global memory												
Warp vote functions												
Double-precision floating-point operations	No			Yes								
Atomic functions operating on 64-bit integer values in shared memory	No				Yes							
Floating-point atomic addition operating on 32-bit words in global and shared memory												
_ballot()												
_threadfence_system()												
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()												
Surface functions												
3D grid of thread block												
Warp shuffle functions	No					Yes						
Funnel shift	No						Yes					
Dynamic parallelism	No							Yes				
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No								Yes			
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No									Yes		
Tensor core	No										Yes	

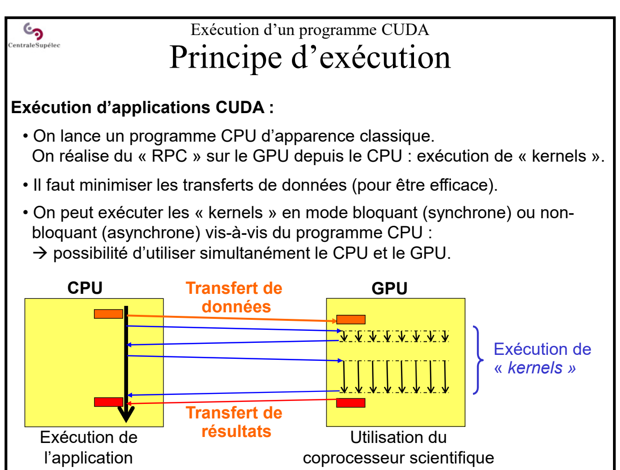
Architecture d'un GPU NVIDIA

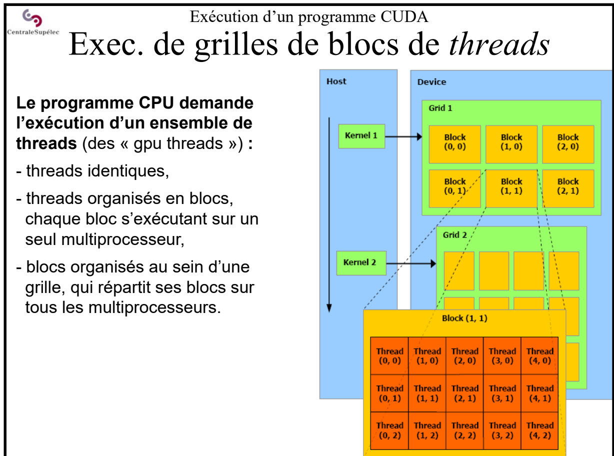
Quelques valeurs (gamme Tesla)

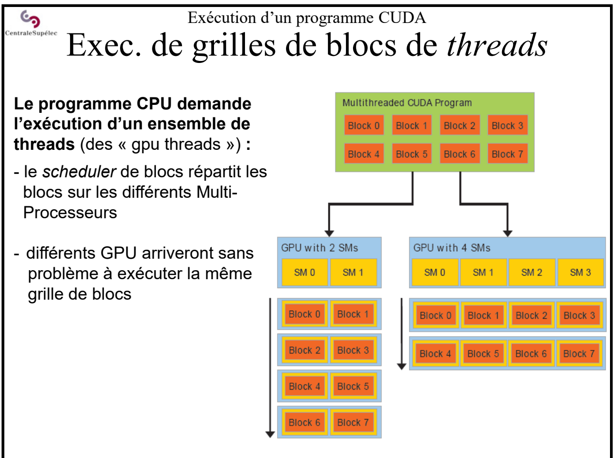
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125

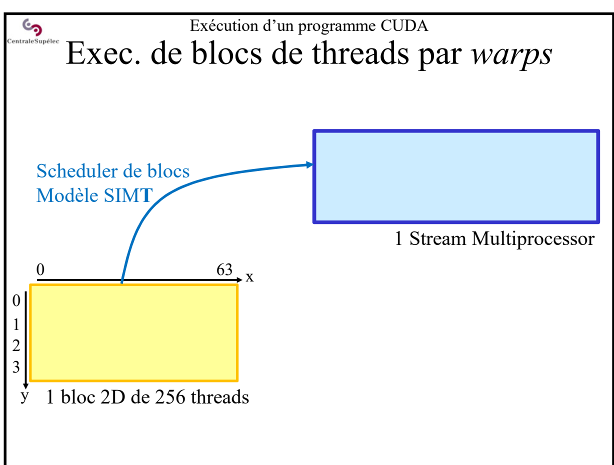
Certaines capacités de calculs n'évoluent pas de manière monotone

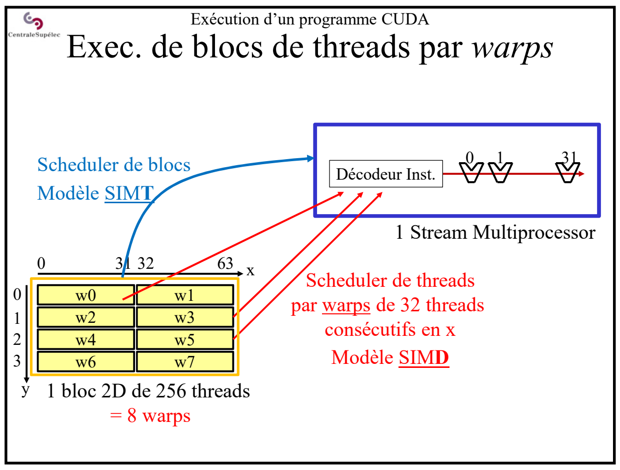

Architecture des GPU et principes de base de CUDA
2 – Exécution d'un pgm CUDA
Principe d'exécution
Exec. de grilles de blocs de threads
Exec. de blocs de threads par warps
Granularité de la grille et des blocs

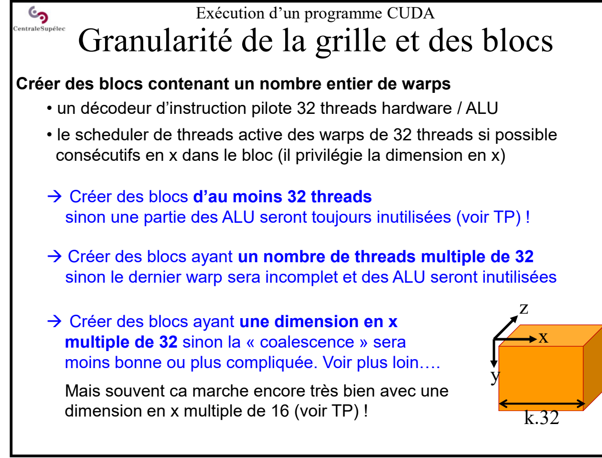

Exécution d'un programme CUDA
Principe d'exécution
Exécution d'applications CUDA :
• On lance un programme CPU d'apparence classique.
On réalise du « RPC » sur le GPU depuis le CPU : exécution de « kernels ».
• Il faut minimiser les transferts de données (pour être efficace).
• On peut exécuter les « kernels » en mode bloquant (synchrone) ou non-bloquant (asynchrone) vis-à-vis du programme CPU :
→ possibilité d'utiliser simultanément le CPU et le GPU.
CPU
Transfert de données
GPU
Exécution de « kernels »
Transfert de résultats
Exécution de l'application
Utilisation du coprocesseur scientifique

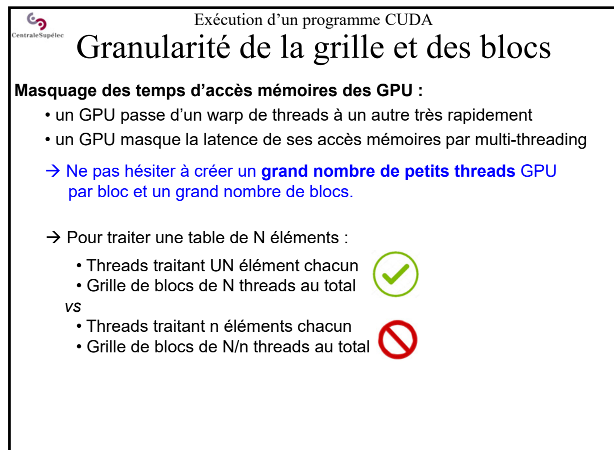

Exécution d'un programme CUDA
Exec. de grilles de blocs de threads
Le programme CPU demande l'exécution d'un ensemble de threads (des « gpu threads ») :
- threads identiques,
- threads organisés en blocs, chaque bloc s'exécutant sur un seul multiprocesseur,
- blocs organisés au sein d'une grille, qui répartit ses blocs sur tous les multiprocesseurs.
Host
Device
Kernel 1
Kernel 2
Grid 1
Grid 2
Block (1, 1)


Exécution d'un programme CUDA
Exec. de grilles de blocs de threads
Le programme CPU demande l'exécution d'un ensemble de threads (des « gpu threads ») :
- le scheduler de blocs répartit les blocs sur les différents Multi-Processeurs
- différents GPU arriveront sans problème à exécuter la même grille de blocs
Multithreaded CUDA Program
GPU with 2 SMs
GPU with 4 SMs


Exécution d'un programme CUDA
Exec. de blocs de threads par warps
Scheduler de blocs
Modèle SIMT
1 Stream Multiprocessor
0
63
x
y
1 bloc 2D de 256 threads

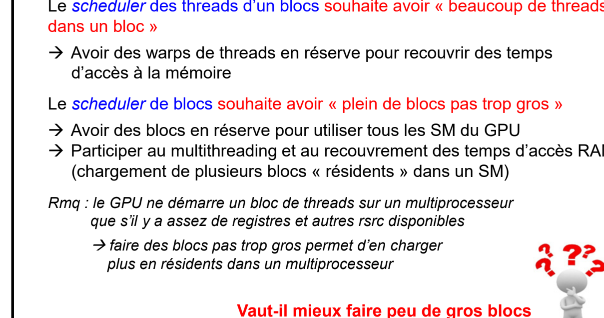

Exécution d'un programme CUDA
Exec. de blocs de threads par warps
Scheduler de blocs
Modèle SIMT
Décodeur Inst.
1 Stream Multiprocessor
w0 w1
w2 w3
w4 w5
w6 w7
Scheduler de threads par warps de 32 threads consécutifs en x
Modèle SIMD
1 bloc 2D de 256 threads
= 8 warps





Exécution d'un programme CUDA

Granularité de la grille et des blocs



Exécution d'un programme CUDA

Granularité de la grille et des blocs

Combien de threads/bloc et de blocs/grille ?

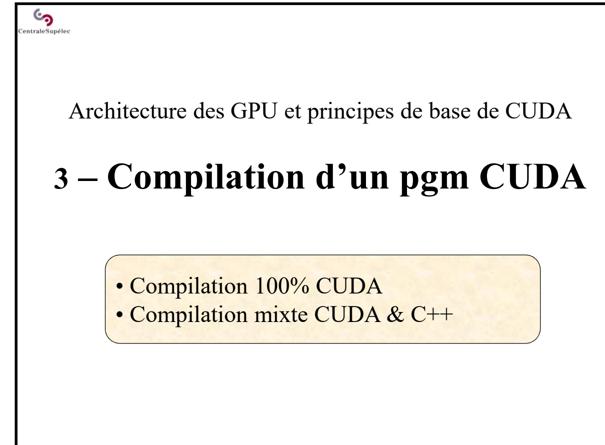
Plusieurs stratégies possibles :

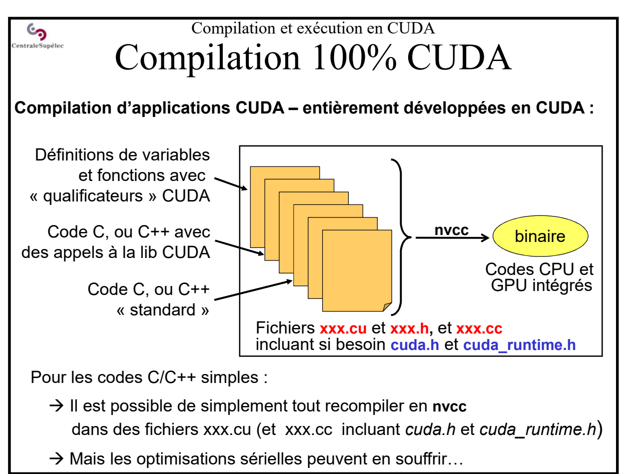
- **Faire des blocs de taille moyenne (128/256 threads) pour permettre aux deux *schedulers* d'optimiser l'exécution**
- **NVIDIA propose un outil pour calculer la taille des blocs menant à l'occupation maximale des ressources du GPU**
- **Expérimenter des tailles de blocs de 32/64/128/256/512/1024 !**

La solution optimale dépend du code CUDA … et du modèle de GPU

→ Trouver la granularité de grille de blocs optimale peut demander quelques expérimentations

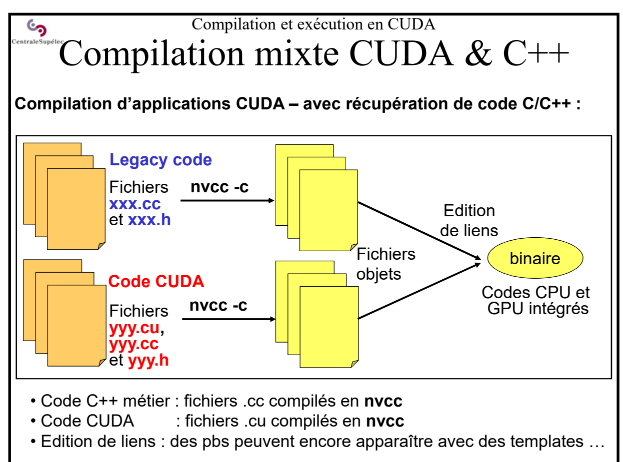
→ Voir TP





Compilation mixte CUDA & C++

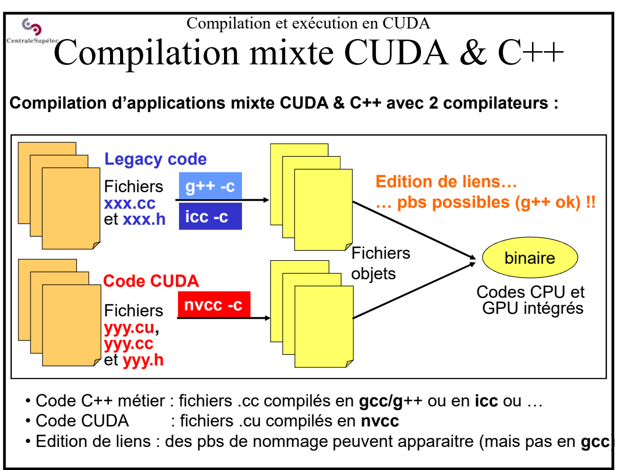
Compilation d'applications CUDA – avec récupération de code C/C++ :



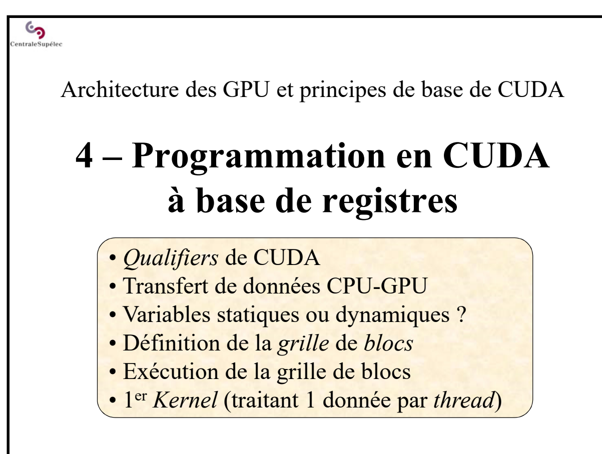
- Code C++ métier : fichiers .cc compilés en **nvcc**
- Code CUDA : fichiers .cu compilés en **nvcc**
- Edition de liens : des pbs peuvent encore apparaître avec des templates …

Compilation mixte CUDA & C++

Compilation d'applications mixte CUDA & C++ avec 2 compilateurs :



- Code C++ métier : fichiers .cc compilés en **gcc/g++** ou en **icc** ou …
- Code CUDA : fichiers .cu compilés en **nvcc**
- Edition de liens : des pbs de nommage peuvent apparaitre (mais pas en **gcc**



« Qualifiers » de CUDA

Fonctionnement des « qualifiers » de CUDA :

	`__device__`	`__host__` (default)	`__global__`
Fonctions	Appel sur GPU Exec sur GPU	Appel sur CPU Exec sur CPU	Appel sur CPU Exec sur GPU
	`__device__`	`__constant__`	`__shared__`
Variables	Mémoire globale GPU Durée de vie de l'application Accessible par les codes GPU et CPU	Mémoire constante GPU Durée de vie de l'application Ecrit par code CPU, lu par code GPU	Mémoire partagée d'un multiprocesseur Durée de vie du *block de threads* Accessible par le code GPU, sert à *cacher* la mémoire globale GPU

→ Les « qualifiers » différencient les parties de code GPU et CPU.

Transfert de données CPU-GPU

Variables allouées sur GPU à la compilation (« *symboles* ») :

```
float TabCPU[N];               // Array on CPU
__device__ float TabGPU[N];    // Array on GPU (symbol)
```

Transfert de données du CPU vers un « *symbole* » stocké sur GPU :

```
// Copy all TabCPU array into TabGPU array
cudaMemcpyToSymbol(TabGPU,&TabCPU[0],
                   sizeof(float)*N,0,
                   cudaMemcpyHostToDevice);

// Copy 2nd half of TabCPU array into 2nd half TabGPU array
cudaMemcpyToSymbol(TabGPU,&TabCPU[N/2],
                   sizeof(float)*N/2,sizeof(float)*N/2,
                   cudaMemcpyHostToDevice);
```

Transfert de données d'un « *symbole* » stocké sur GPU vers le CPU :

```
// Copy all TabGPU array into TabCPU array
cudaMemcpyFromSymbol(&TabCPU[0],TabGPU,
                     sizeof(float)*N,0,
                     cudaMemcpyDeviceToHost);
```


Transfert de données CPU-GPU

Variables allouées sur GPU à l'exécution :

```
float *TabCPU;          // Dynamic array on CPU
float *TabGPU;          // Dynamic array on GPU
cudaError_t cudaStat;   // Result of op on dynamic CUDA vars.

// Allocation of the dynamic arrays from the CPU
TabCPU = (float *) malloc(N*sizeof(float));
cudaStat = cudaMalloc((void **) &TabGPU, N*sizeof(float));
```

Copie de variables dynamiques (CPU → GPU) :

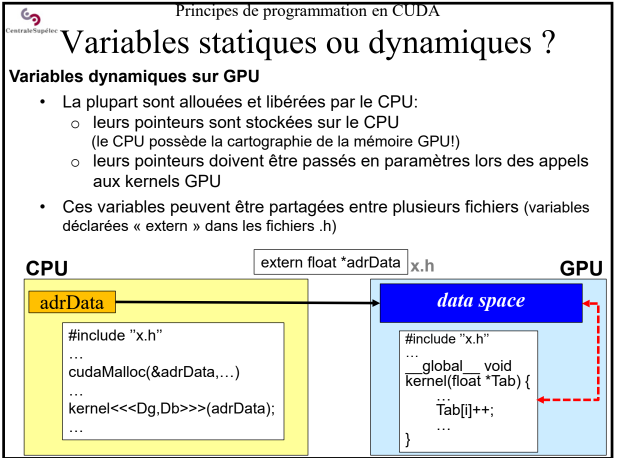
```
// Copy TabCPU dynamic array into TabGPU dynamic array
cudaStat = cudaMemcpy(TabGPU, TabCPU, sizeof(float)*N,
                      cudaMemcpyHostToDevice);
```

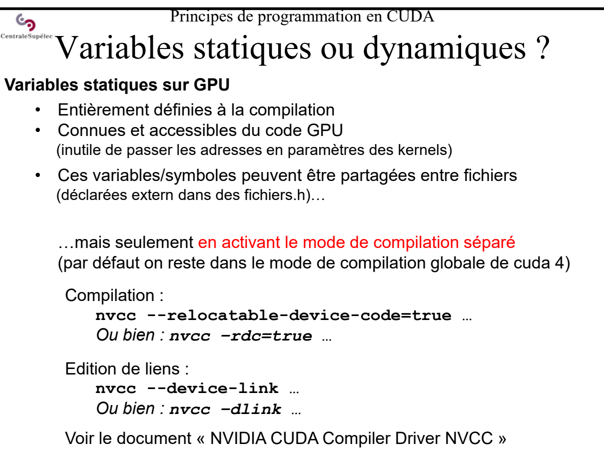
Copie de variables dynamiques (GPU → CPU) :

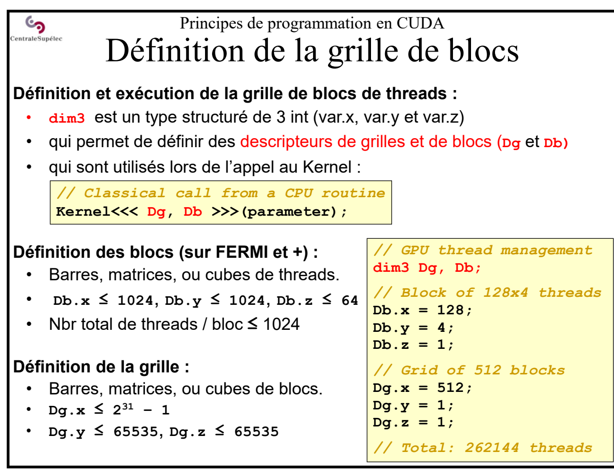
```
// Copy TabCPU dynamic array into TabGPU dynamic array
cudaStat = cudaMemcpy(TabCPU, TabGPU, sizeof(float)*N,
                      cudaMemcpyDeviceToHost);
```

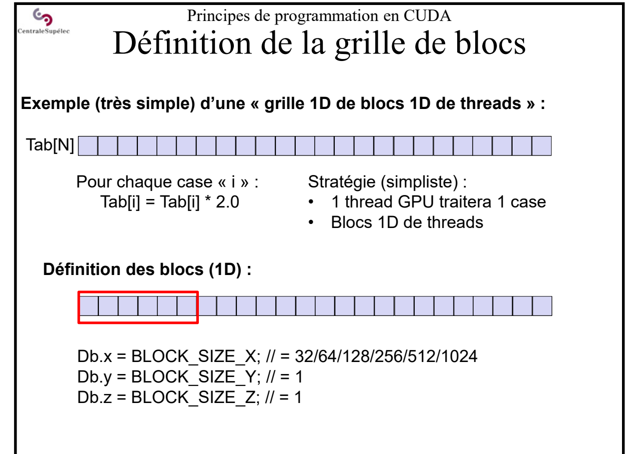
Libération des allocations dynamiques

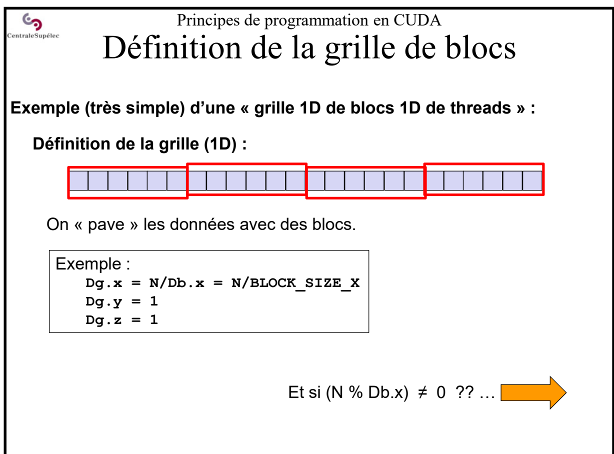
```
free(TabCPU);
cudaStat = cudaFree(TabGPU);
```

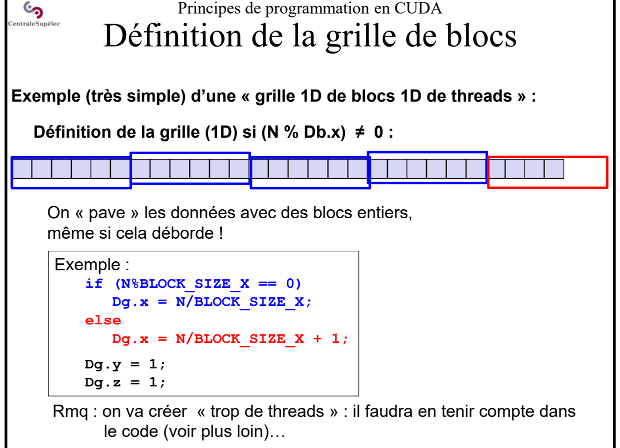

Principes de programmation en CUDA
Variables statiques ou dynamiques ?
Variables dynamiques sur GPU
La plupart sont allouées et libérées par le CPU:
leurs pointeurs sont stockées sur le CPU
(le CPU possède la cartographie de la mémoire GPU!)
leurs pointeurs doivent être passés en paramètres lors des appels aux kernels GPU
Ces variables peuvent être partagées entre plusieurs fichiers (variables déclarées « extern » dans les fichiers .h)
CPU
extern float *adrData
x.h
GPU
adrData
data space
#include "x.h"
…
cudaMalloc(&adrData,…)
…
kernel<<<Dg,Db>>>(adrData);
…
#include "x.h"
…
__global__ void
kernel(float *Tab) {
…
Tab[i]++;
…
}

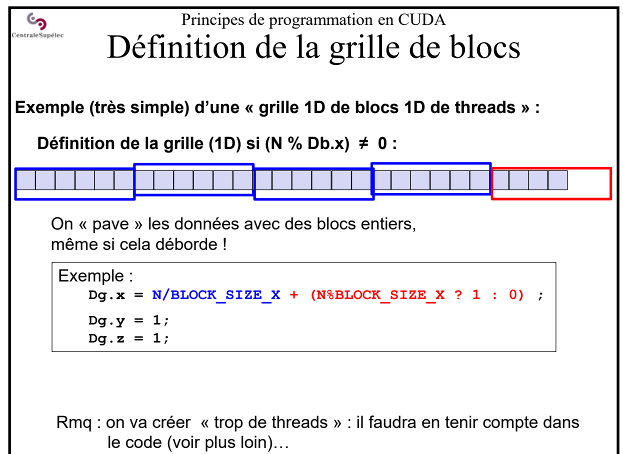

Principes de programmation en CUDA
Variables statiques ou dynamiques ?
Variables statiques sur GPU
Entièrement définies à la compilation
Connues et accessibles du code GPU
(inutile de passer les adresses en paramètres des kernels)
Ces variables/symboles peuvent être partagées entre fichiers
(déclarées extern dans des fichiers.h)…
…mais seulement en activant le mode de compilation séparé
(par défaut on reste dans le mode de compilation globale de cuda 4)
Compilation :
nvcc --relocatable-device-code=true …
Ou bien : nvcc -rdc=true …
Edition de liens :
nvcc --device-link …
Ou bien : nvcc -dlink …
Voir le document « NVIDIA CUDA Compiler Driver NVCC »

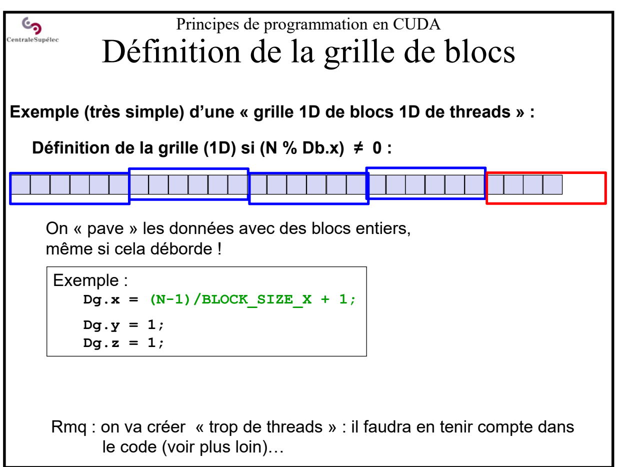

Principes de programmation en CUDA
Définition de la grille de blocs
Définition et exécution de la grille de blocs de threads :
dim3 est un type structuré de 3 int (var.x, var.y et var.z)
qui permet de définir des descripteurs de grilles et de blocs (Dg et Db)
qui sont utilisés lors de l'appel au Kernel :
// Classical call from a CPU routine
Kernel<<< Dg, Db >>>(parameter);
Définition des blocs (sur FERMI et +) :
Barres, matrices, ou cubes de threads.
Db.x ≤ 1024, Db.y ≤ 1024, Db.z ≤ 64
Nbr total de threads / bloc ≤ 1024
Définition de la grille :
Barres, matrices, ou cubes de blocs.
Dg.x ≤ 2³¹ – 1
Dg.y ≤ 65535, Dg.z ≤ 65535
// GPU thread management
dim3 Dg, Db;
// Block of 128x4 threads
Db.x = 128;
Db.y = 4;
Db.z = 1;
// Grid of 512 blocks
Dg.x = 512;
Dg.y = 1;
Dg.z = 1;
// Total: 262144 threads

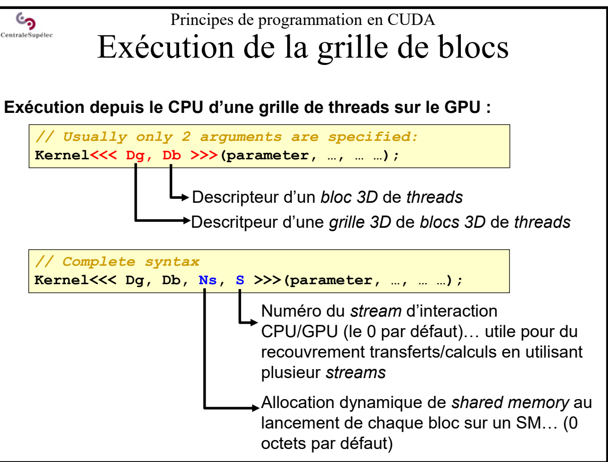

Principes de programmation en CUDA
Définition de la grille de blocs
Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :
Tab[N]
Pour chaque case « i » :
Tab[i] = Tab[i] * 2.0
Stratégie (simpliste) :
1 thread GPU traitera 1 case
Blocs 1D de threads
Définition des blocs (1D) :
Db.x = BLOCK_SIZE_X; // = 32/64/128/256/512/1024
Db.y = BLOCK_SIZE_Y; // = 1
Db.z = BLOCK_SIZE_Z; // = 1

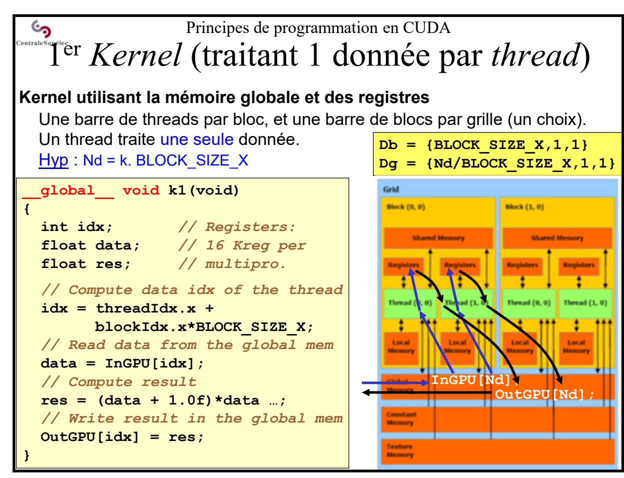

Principes de programmation en CUDA
Définition de la grille de blocs
Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :
Définition de la grille (1D) :
On « pave » les données avec des blocs.
Exemple :
Dg.x = N/Db.x = N/BLOCK_SIZE_X
Dg.y = 1
Dg.z = 1
Et si (N % Db.x) ≠ 0 ?? …

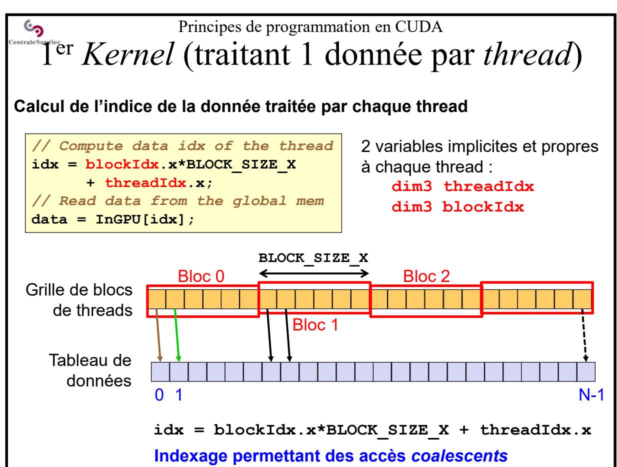

Principes de programmation en CUDA
Définition de la grille de blocs
Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :
Définition de la grille (1D) si (N % Db.x) ≠ 0 :
On « pave » les données avec des blocs entiers,
même si cela déborde !
Exemple :
if (N%BLOCK_SIZE_X == 0)
Dg.x = N/BLOCK_SIZE_X;
else
Dg.x = N/BLOCK_SIZE_X + 1;
Dg.y = 1;
Dg.z = 1;
Rmq : on va créer « trop de threads » : il faudra en tenir compte dans
le code (voir plus loin)…

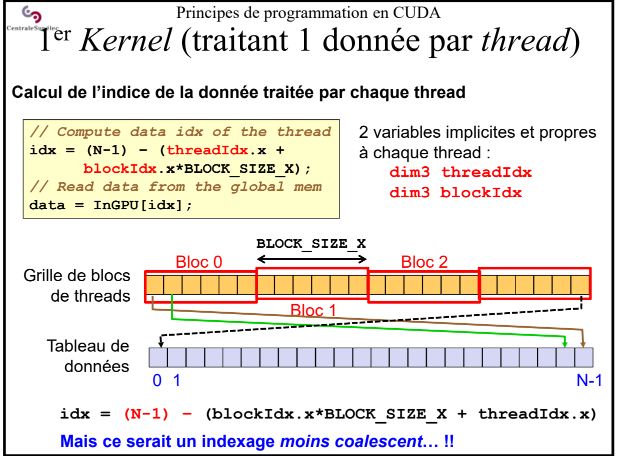

Principes de programmation en CUDA
Définition de la grille de blocs
Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :
Définition de la grille (1D) si (N % Db.x) ≠ 0 :
On « pave » les données avec des blocs entiers,
même si cela déborde !
Exemple :
Dg.x = N/BLOCK_SIZE_X + (N%BLOCK_SIZE_X ? 1 : 0) ;
Dg.y = 1;
Dg.z = 1;
Rmq : on va créer « trop de threads » : il faudra en tenir compte dans
le code (voir plus loin)…

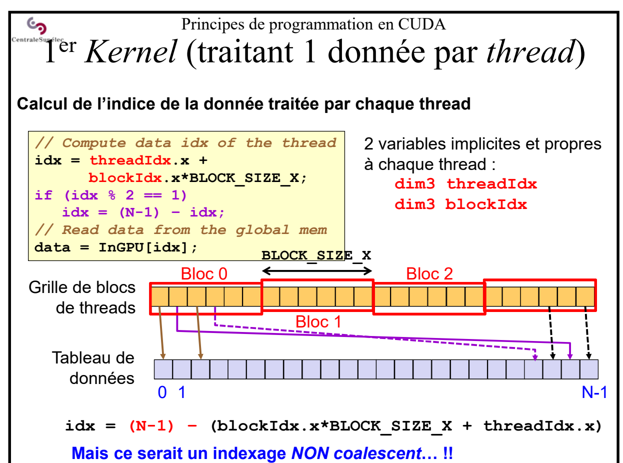

Principes de programmation en CUDA
Définition de la grille de blocs
Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :
Définition de la grille (1D) si (N % Db.x) ≠ 0 :
On « pave » les données avec des blocs entiers,
même si cela déborde !
Exemple :
Dg.x = (N-1)/BLOCK_SIZE_X + 1;
Dg.y = 1;
Dg.z = 1;
Rmq : on va créer « trop de threads » : il faudra en tenir compte dans
le code (voir plus loin)…

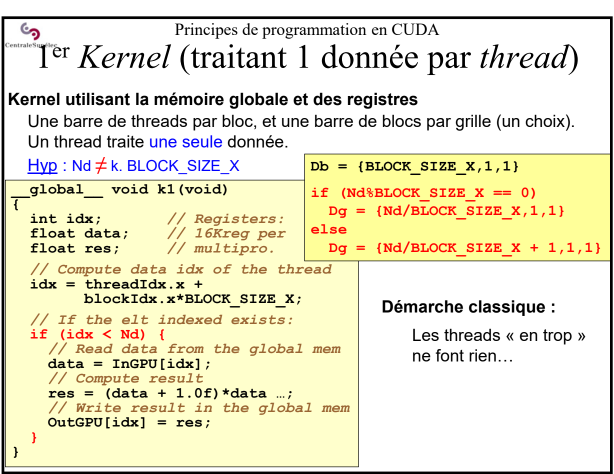

Principes de programmation en CUDA
Exécution de la grille de blocs
Exécution depuis le CPU d'une grille de threads sur le GPU :
// Usually only 2 arguments are specified:
Kernel<<< Dg, Db >>>(parameter, …, … …);
Descripteur d'un bloc 3D de threads
Descritpeur d'une grille 3D de blocs 3D de threads
// Complete syntax
Kernel<<< Dg, Db, Ns, S >>>(parameter, …, … …);
Numéro du stream d'interaction
CPU/GPU (le 0 par défaut)… utile pour du
recouvrement transferts/calculs en utilisant
plusieur streams
Allocation dynamique de shared memory au
lancement de chaque bloc sur un SM… (0
octets par défaut)


Principes de programmation en CUDA
1er Kernel (traitant 1 donnée par thread)
Kernel utilisant la mémoire globale et des registres
Une barre de threads par bloc, et une barre de blocs par grille (un choix).
Un thread traite une seule donnée.
Hyp : Nd = k. BLOCK_SIZE_X
Db = {BLOCK_SIZE_X,1,1}
Dg = {Nd/BLOCK_SIZE_X,1,1}
__global__ void k1(void)
{
int idx; // Registers:
float data; // 16 Kreg per
float res; // multipro.
// Compute data idx of the thread
idx = threadIdx.x +
blockIdx.x*BLOCK_SIZE_X;
// Read data from the global mem
data = InGPU[idx];
// Compute result
res = (data + 1.0f)*data …;
// Write result in the global mem
OutGPU[idx] = res;
}
InGPU[Nd]
OutGPU[Nd];


Principes de programmation en CUDA
1er Kernel (traitant 1 donnée par thread)
Calcul de l'indice de la donnée traitée par chaque thread
// Compute data idx of the thread
idx = blockIdx.x*BLOCK_SIZE_X
+ threadIdx.x;
// Read data from the global mem
data = InGPU[idx];
2 variables implicites et propres
à chaque thread :
dim3 threadIdx
dim3 blockIdx
BLOCK_SIZE_X
Bloc 0
Bloc 1
Bloc 2
Grille de blocs
de threads
Tableau de
données
0 1
N-1
idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x
Indexage permettant des accès coalescents


Principes de programmation en CUDA
1er Kernel (traitant 1 donnée par thread)
Calcul de l'indice de la donnée traitée par chaque thread
// Compute data idx of the thread
idx = (N-1) – (threadIdx.x +
blockIdx.x*BLOCK_SIZE_X);
// Read data from the global mem
data = InGPU[idx];
2 variables implicites et propres
à chaque thread :
dim3 threadIdx
dim3 blockIdx
BLOCK_SIZE_X
Bloc 0
Bloc 1
Bloc 2
Grille de blocs
de threads
Tableau de
données
0 1
N-1
idx = (N-1) – (blockIdx.x*BLOCK_SIZE_X + threadIdx.x)
Mais ce serait un indexage moins coalescent… !!

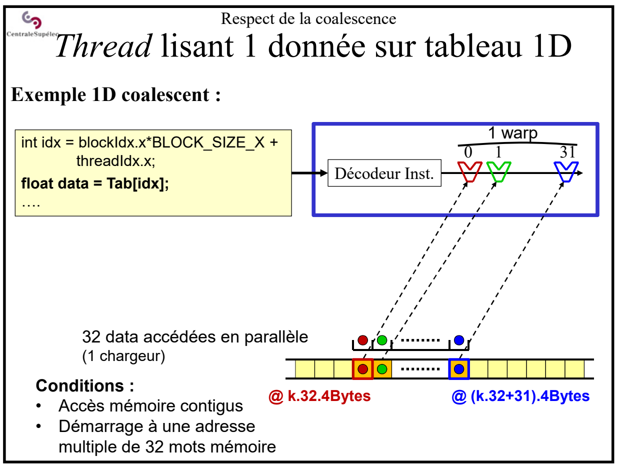


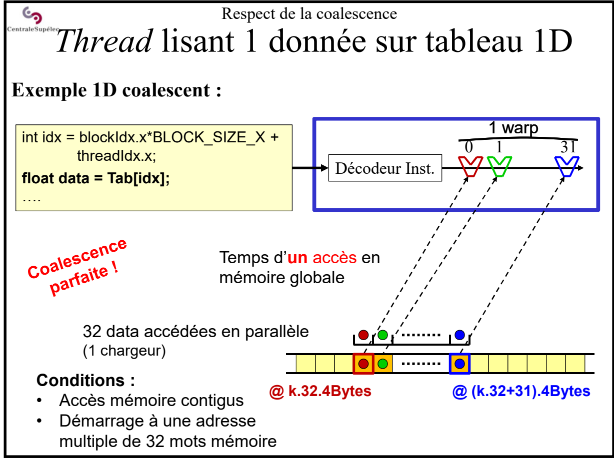


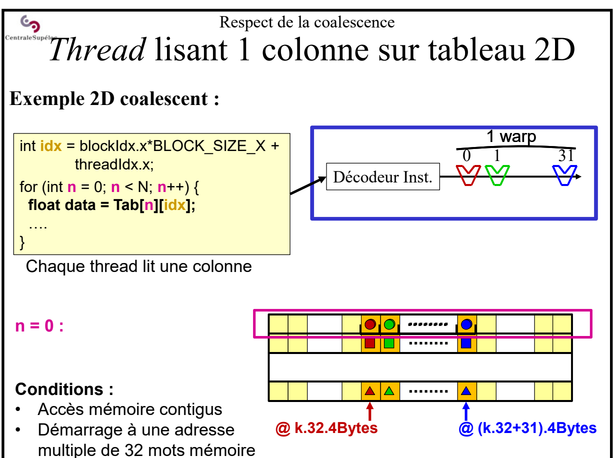
Architecture des GPU et principes de base de CUDA

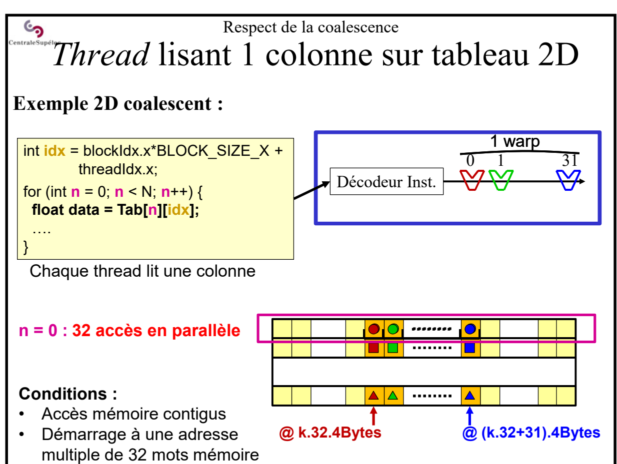
5 – Respect de la « coalescence »

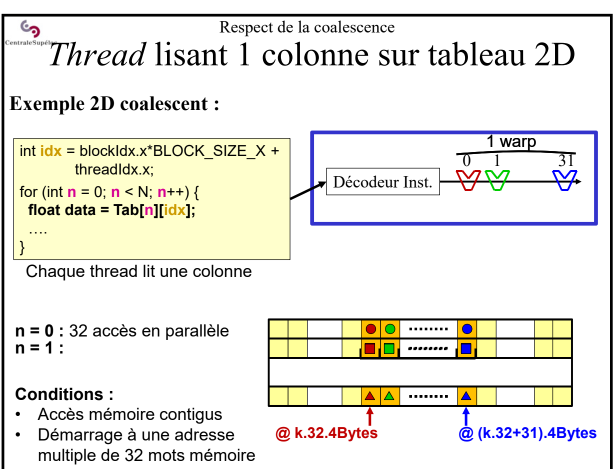
- *Thread* lisant 1 donnée sur tableau 1D
- *Thread* lisant 1 colonne sur tableau 2D
- *Thread* lisant 1 ligne sur tableau 2D
- *Thread* lisant *n* données sur tableau 1D
- Sensibilité de la coalescence
- Règles de développement

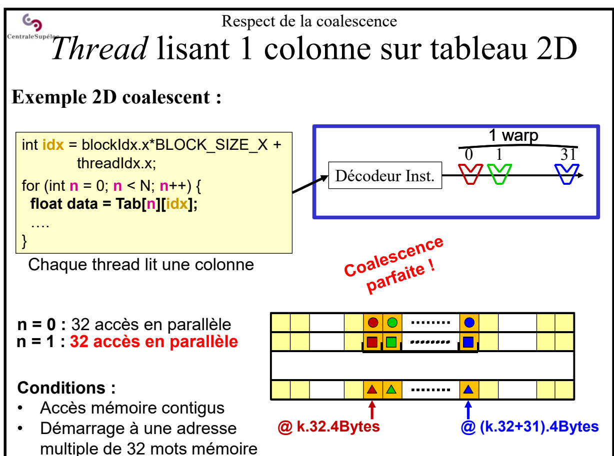


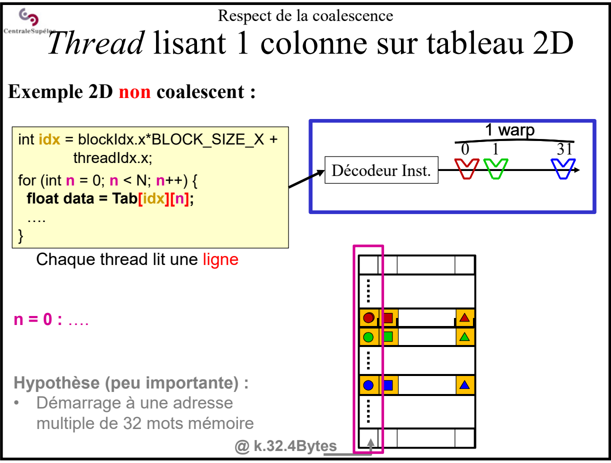


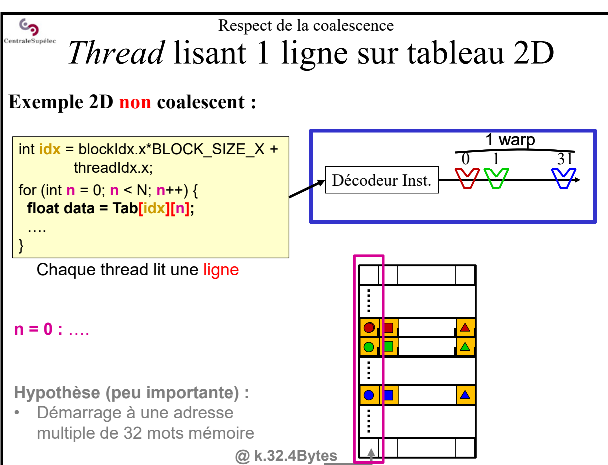


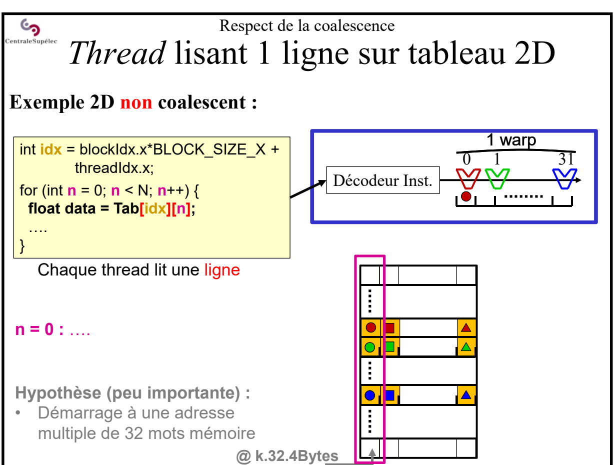

Respect de la coalescence
Thread lisant 1 colonne sur tableau 2D
Exemple 2D coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[n][idx];
....
}
Chaque thread lit une colonne
1 warp
Décodeur Inst.
0 1 31
n = 0 : 32 accès en parallèle
Conditions :
Accès mémoire contigus
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes
@ (k.32+31).4Bytes

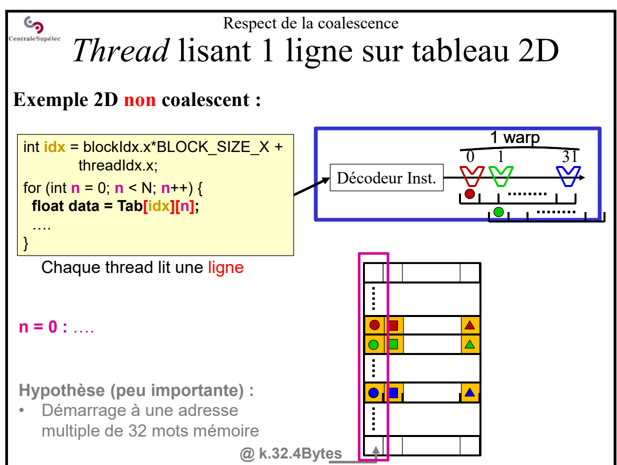

Respect de la coalescence
Thread lisant 1 colonne sur tableau 2D
Exemple 2D coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[n][idx];
....
}
Chaque thread lit une colonne
1 warp
Décodeur Inst.
0 1 31
n = 0 : 32 accès en parallèle
n = 1 :
Conditions :
Accès mémoire contigus
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes
@ (k.32+31).4Bytes

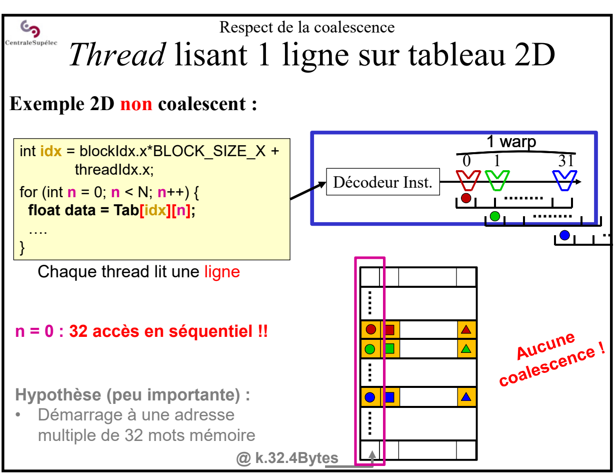

Respect de la coalescence
Thread lisant 1 colonne sur tableau 2D
Exemple 2D coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[n][idx];
....
}
Chaque thread lit une colonne
1 warp
Décodeur Inst.
0 1 31
Coalescence parfaite !
n = 0 : 32 accès en parallèle
n = 1 : 32 accès en parallèle
Conditions :
Accès mémoire contigus
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes
@ (k.32+31).4Bytes

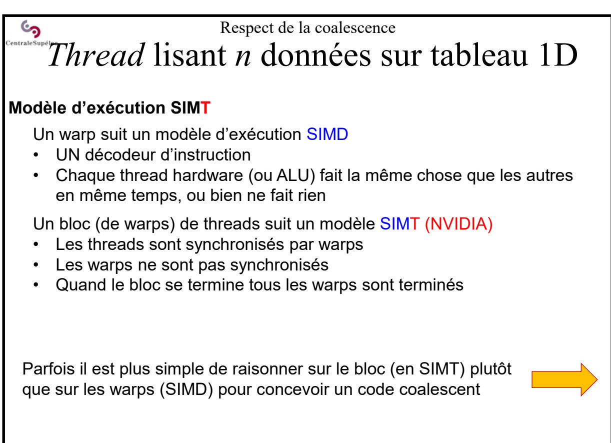

Respect de la coalescence
Thread lisant 1 colonne sur tableau 2D
Exemple 2D non coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[idx][n];
....
}
Chaque thread lit une ligne
1 warp
Décodeur Inst.
0 1 31
n = 0 : ....
Hypothèse (peu importante) :
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes

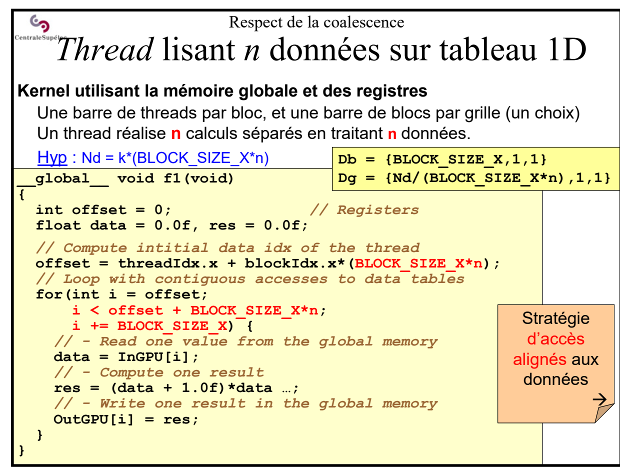

Respect de la coalescence
Thread lisant 1 ligne sur tableau 2D
Exemple 2D non coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[idx][n];
....
}
Chaque thread lit une ligne
1 warp
Décodeur Inst.
0 1 31
n = 0 : ....
Hypothèse (peu importante) :
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes

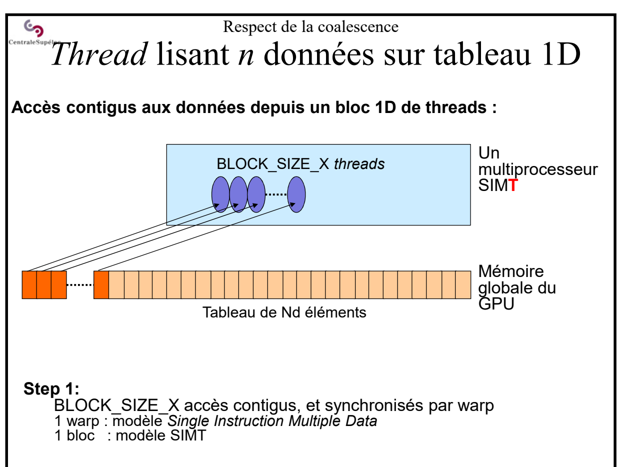

Respect de la coalescence
Thread lisant 1 ligne sur tableau 2D
Exemple 2D non coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[idx][n];
....
}
Chaque thread lit une ligne
1 warp
Décodeur Inst.
0 1 31
n = 0 : ....
Hypothèse (peu importante) :
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes

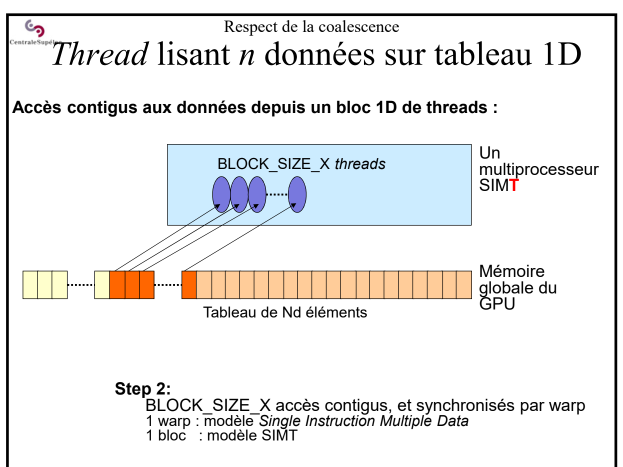

Respect de la coalescence
Thread lisant 1 ligne sur tableau 2D
Exemple 2D non coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[idx][n];
....
}
Chaque thread lit une ligne
Décodeur Inst.
1 warp
0 1 31
n = 0 : ....
Hypothèse (peu importante) :
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes

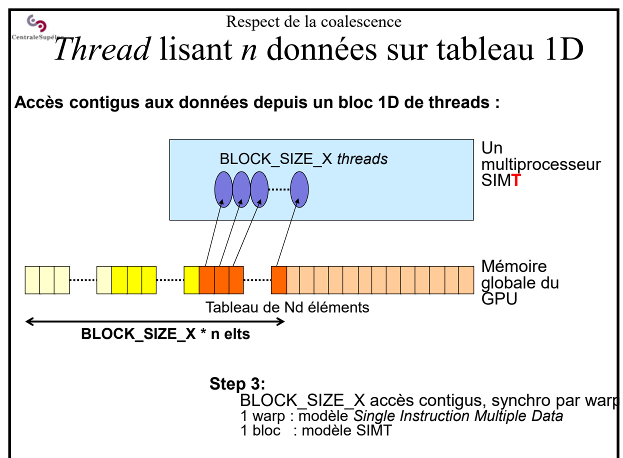

Respect de la coalescence
Thread lisant 1 ligne sur tableau 2D
Exemple 2D non coalescent :
int idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x;
for (int n = 0; n < N; n++) {
float data = Tab[idx][n];
....
}
Chaque thread lit une ligne
Décodeur Inst.
1 warp
0 1 31
n = 0 : 32 accès en séquentiel !!
Aucune coalescence !
Hypothèse (peu importante) :
Démarrage à une adresse multiple de 32 mots mémoire
@ k.32.4Bytes

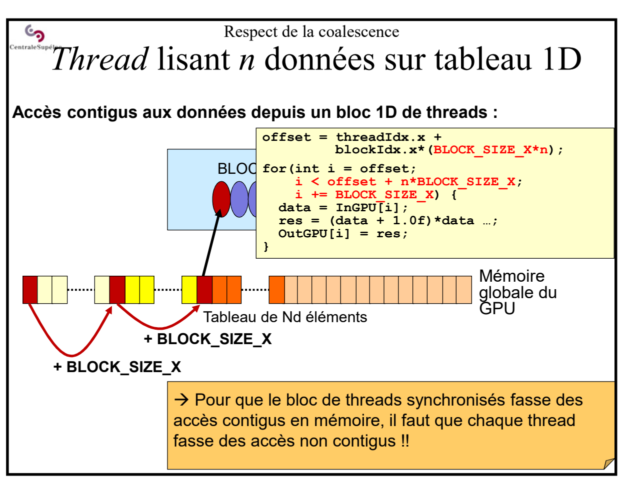

Respect de la coalescence
Thread lisant n données sur tableau 1D
Modèle d'exécution SIMT
Un warp suit un modèle d'exécution SIMD
UN décodeur d'instruction
Chaque thread hardware (ou ALU) fait la même chose que les autres en même temps, ou bien ne fait rien
Un bloc (de warps) de threads suit un modèle SIMT (NVIDIA)
Les threads sont synchronisés par warps
Les warps ne sont pas synchronisés
Quand le bloc se termine tous les warps sont terminés
Parfois il est plus simple de raisonner sur le bloc (en SIMT) plutôt que sur les warps (SIMD) pour concevoir un code coalescent

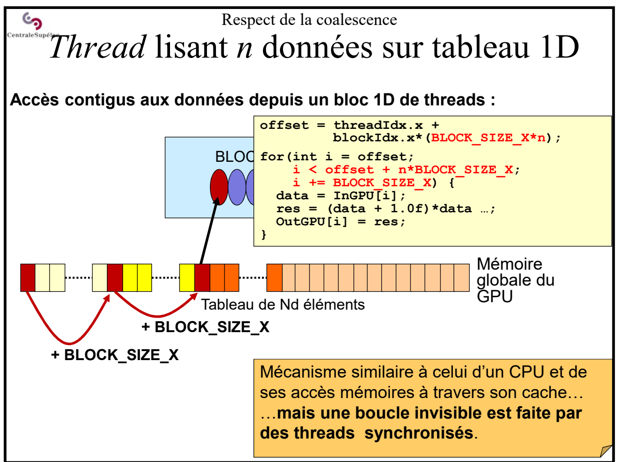

Respect de la coalescence
Thread lisant n données sur tableau 1D
Kernel utilisant la mémoire globale et des registres
Une barre de threads par bloc, et une barre de blocs par grille (un choix)
Un thread réalise n calculs séparés en traitant n données.
Hyp : Nd = k*(BLOCK_SIZE_X*n)
Db = {BLOCK_SIZE_X,1,1}
Dg = {Nd/(BLOCK_SIZE_X*n),1,1}
__global__ void f1(void)
{
int offset = 0; // Registers
float data = 0.0f, res = 0.0f;
// Compute intitial data idx of the thread
offset = threadIdx.x + blockIdx.x*(BLOCK_SIZE_X*n);
// Loop with contiguous accesses to data tables
for(int i = offset;
i < offset + BLOCK_SIZE_X*n;
i += BLOCK_SIZE_X) {
// - Read one value from the global memory
data = InGPU[i];
// - Compute one result
res = (data + 1.0f)*data ...;
// - Write one result in the global memory
OutGPU[i] = res;
}
}
Stratégie d'accès alignés aux données

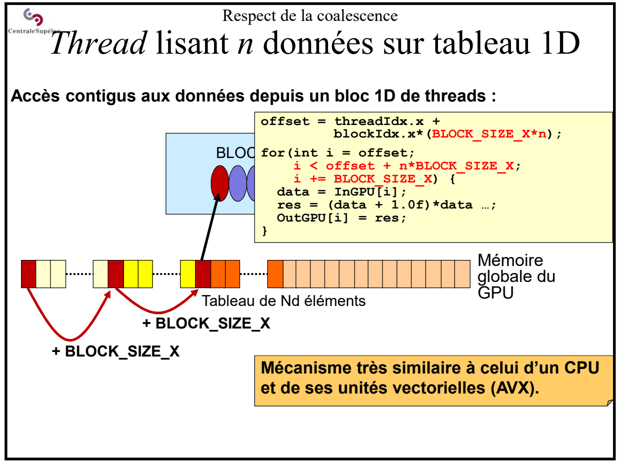

Respect de la coalescence
Thread lisant n données sur tableau 1D
Accès contigus aux données depuis un bloc 1D de threads :
BLOCK_SIZE_X threads
Un multiprocesseur SIMT
Mémoire globale du GPU
Tableau de Nd éléments
Step 1:
BLOCK_SIZE_X accès contigus, et synchronisés par warp
1 warp : modèle Single Instruction Multiple Data
1 bloc : modèle SIMT

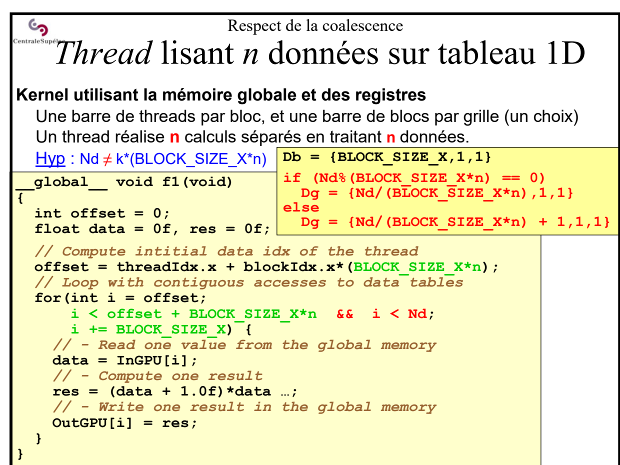

Respect de la coalescence
Thread lisant n données sur tableau 1D
Accès contigus aux données depuis un bloc 1D de threads :
BLOCK_SIZE_X threads
Un multiprocesseur SIMT
Mémoire globale du GPU
Tableau de Nd éléments
Step 2:
BLOCK_SIZE_X accès contigus, et synchronisés par warp
1 warp : modèle Single Instruction Multiple Data
1 bloc : modèle SIMT

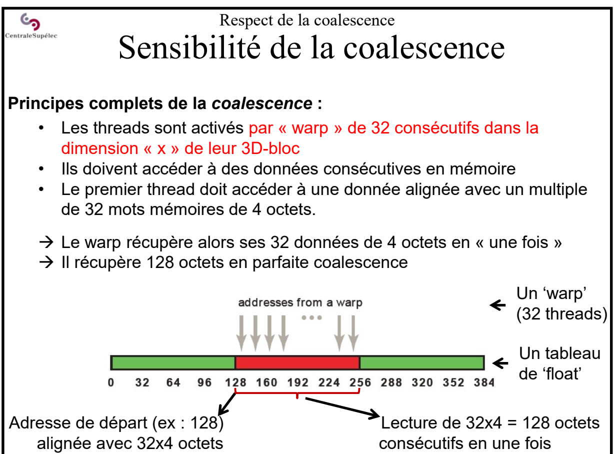

Respect de la coalescence
Thread lisant n données sur tableau 1D
Accès contigus aux données depuis un bloc 1D de threads :
BLOCK_SIZE_X threads
Un multiprocesseur SIMT
Mémoire globale du GPU
Tableau de Nd éléments
BLOCK_SIZE_X * n elts
Step 3:
BLOCK_SIZE_X accès contigus, synchro par warp
1 warp : modèle Single Instruction Multiple Data
1 bloc : modèle SIMT

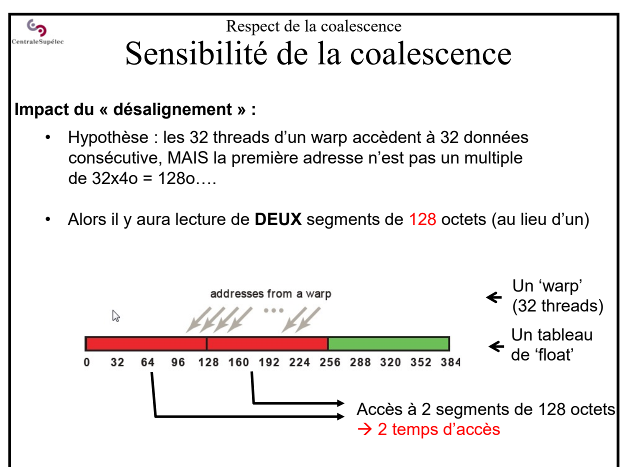

Respect de la coalescence
Thread lisant n données sur tableau 1D
Accès contigus aux données depuis un bloc 1D de threads :
offset = threadIdx.x +
blockIdx.x*(BLOCK_SIZE_X*n);
for(int i = offset;
i < offset + n*BLOCK_SIZE_X;
i += BLOCK_SIZE_X) {
data = InGPU[i];
res = (data + 1.0f)*data …;
OutGPU[i] = res;
}
Mémoire globale du GPU
Tableau de Nd éléments
+ BLOCK_SIZE_X
+ BLOCK_SIZE_X
→ Pour que le bloc de threads synchronisés fasse des accès contigus en mémoire, il faut que chaque thread fasse des accès non contigus !!

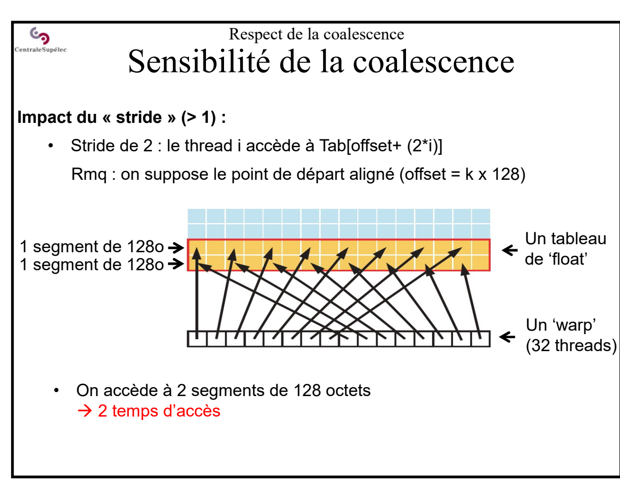

Respect de la coalescence
Thread lisant n données sur tableau 1D
Accès contigus aux données depuis un bloc 1D de threads :
offset = threadIdx.x +
blockIdx.x*(BLOCK_SIZE_X*n);
for(int i = offset;
i < offset + n*BLOCK_SIZE_X;
i += BLOCK_SIZE_X) {
data = InGPU[i];
res = (data + 1.0f)*data …;
OutGPU[i] = res;
}
Mémoire globale du GPU
Tableau de Nd éléments
+ BLOCK_SIZE_X
+ BLOCK_SIZE_X
Mécanisme similaire à celui d'un CPU et de ses accès mémoires à travers son cache…
…mais une boucle invisible est faite par des threads synchronisés.

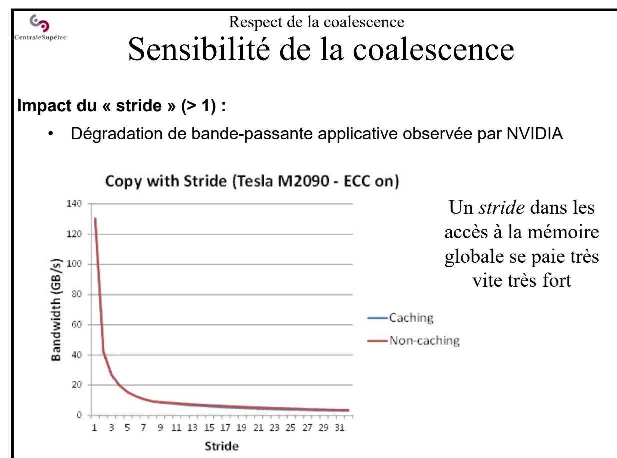

Respect de la coalescence
Thread lisant n données sur tableau 1D
Accès contigus aux données depuis un bloc 1D de threads :
offset = threadIdx.x +
blockIdx.x*(BLOCK_SIZE_X*n);
for(int i = offset;
i < offset + n*BLOCK_SIZE_X;
i += BLOCK_SIZE_X) {
data = InGPU[i];
res = (data + 1.0f)*data …;
OutGPU[i] = res;
}
Mémoire globale du GPU
Tableau de Nd éléments
+ BLOCK_SIZE_X
+ BLOCK_SIZE_X
Mécanisme très similaire à celui d'un CPU et de ses unités vectorielles (AVX).


Respect de la coalescence
Thread lisant n données sur tableau 1D
Kernel utilisant la mémoire globale et des registres
Une barre de threads par bloc, et une barre de blocs par grille (un choix)
Un thread réalise n calculs séparés en traitant n données.
Hyp : Nd ≠ k*(BLOCK_SIZE_X*n)
Db = {BLOCK_SIZE_X,1,1}
if (Nd%(BLOCK_SIZE_X*n) == 0)
Dg = {Nd/(BLOCK_SIZE_X*n),1,1}
else
Dg = {Nd/(BLOCK_SIZE_X*n) + 1,1,1}
__global__ void f1(void)
{
int offset = 0;
float data = 0f, res = 0f;
// Compute intitial data idx of the thread
offset = threadIdx.x + blockIdx.x*(BLOCK_SIZE_X*n);
// Loop with contiguous accesses to data tables
for(int i = offset;
i < offset + BLOCK_SIZE_X*n && i < Nd;
i += BLOCK_SIZE_X) {
// - Read one value from the global memory
data = InGPU[i];
// - Compute one result
res = (data + 1.0f)*data …;
// - Write one result in the global memory
OutGPU[i] = res;
}
}


Respect de la coalescence
Sensibilité de la coalescence
Principes complets de la coalescence :
Les threads sont activés par « warp » de 32 consécutifs dans la dimension « x » de leur 3D-bloc
Ils doivent accéder à des données consécutives en mémoire
Le premier thread doit accéder à une donnée alignée avec un multiple de 32 mots mémoires de 4 octets.
→ Le warp récupère alors ses 32 données de 4 octets en « une fois »
→ Il récupère 128 octets en parfaite coalescence
addresses from a warp
Un 'warp' (32 threads)
Un tableau de 'float'
0 32 64 96 128 160 192 224 256 288 320 352 384
Adresse de départ (ex : 128) alignée avec 32x4 octets
Lecture de 32x4 = 128 octets consécutifs en une fois







Règles de développement

Mise en place de la coalescence :

- La coalescence reste le premier « souci » du développement en CUDA
- Les caches améliorent les performances mais ne masquent pas les défauts de coalescences

→ **Il faut soigner la coalescence dès la conception de l'algorithme**

→ Il faut concevoir un ensemble **'stockage des données et accès'** qui vérifie la coalescence

Utiliser la 'shared memory' peut aider (voir plus loin), car on écrit un algorithme de cache dédié au problème.

Règles de développement

Vérifier la coalescence lors de la conception :

Démarche :

- **Considérer deux threads successifs en X dans un warp** (leur *threadIdx.x* sont des entiers successifs)
- **Identifier leurs accès en mémoire globale**

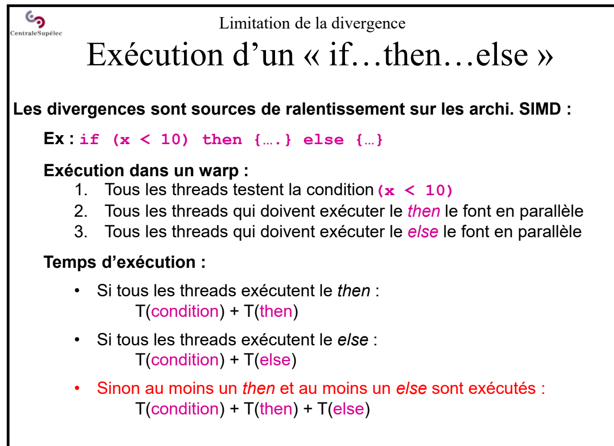
Analyse :

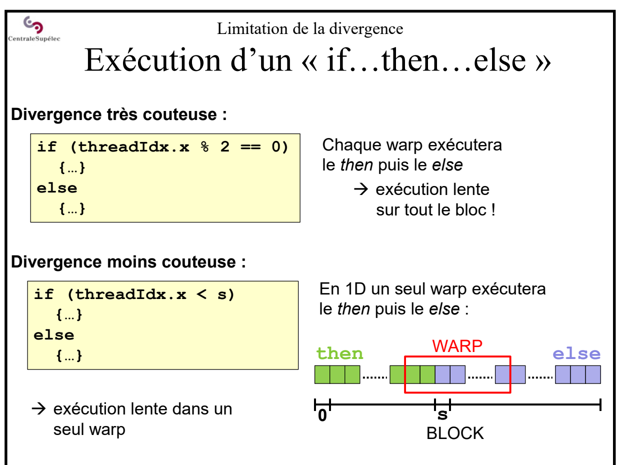
- Si le second accède à la case suivante du premier lors de chaque accès, alors : très bonne coalescence
- Si les deux accèdent à la même case : acceptable (gaspillage de Bw mémoire, mais un seul accès mémoire)
- Si les 32 threads du warp accèdent globalement à des données situées dans le même segment de 32 mots mémoire : bonne coalescence (supportée par les GPU modernes)
- Sinon … revoir les structures de données et les codes de calculs pour améliorer la coalescence.

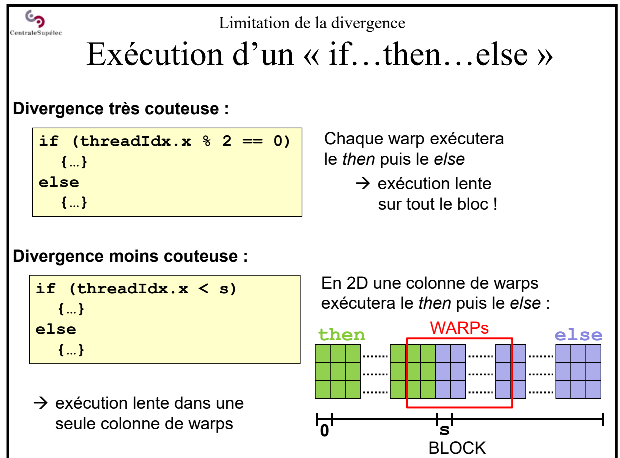
Architecture des GPU et principes de base de CUDA

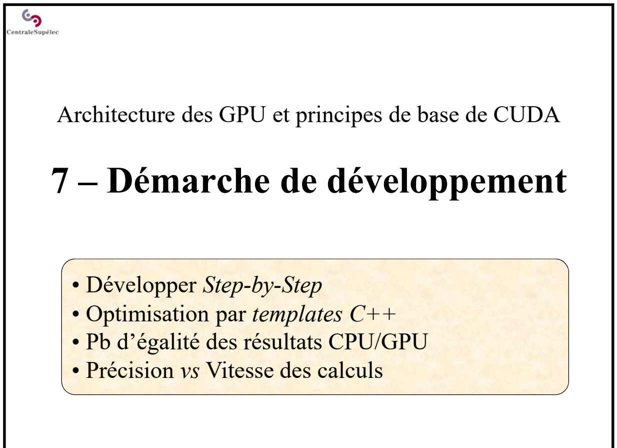
6 – Limitation de la « divergence »

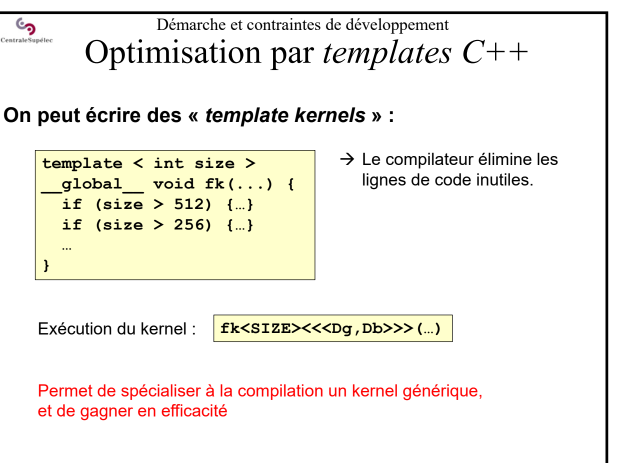
- *Kernel* traitant 1 donnée par *thread*
- *Kernel* traitant n données par *thread*
- Principes d'accès *coalescent* aux données











Démarche et contraintes de développement

Optimisation par *templates C++*

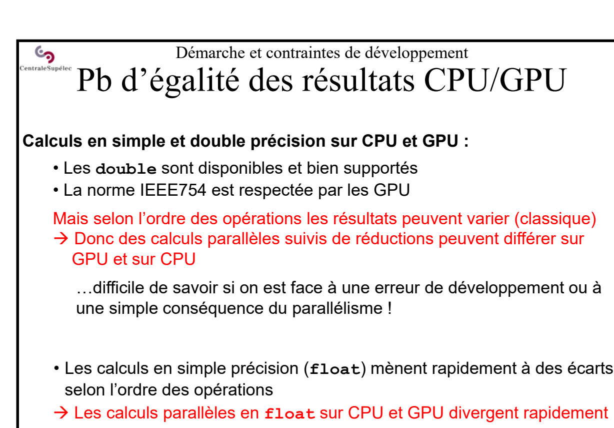
On peut écrire des « *template kernels* » :

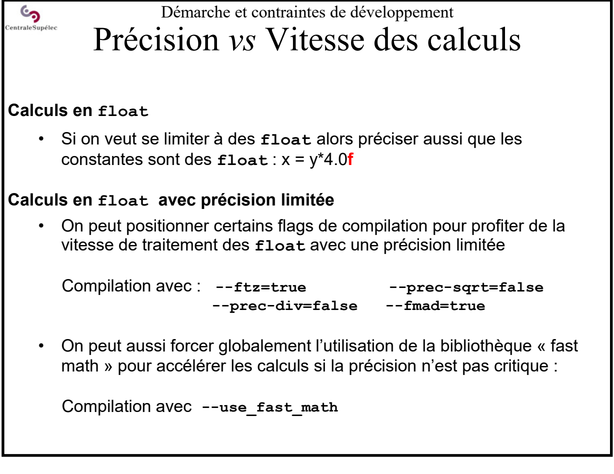
```
template < int size >
__global__ void fk(...) {
  if (size > 512) {…}
  if (size > 256) {…}
  …
}
```

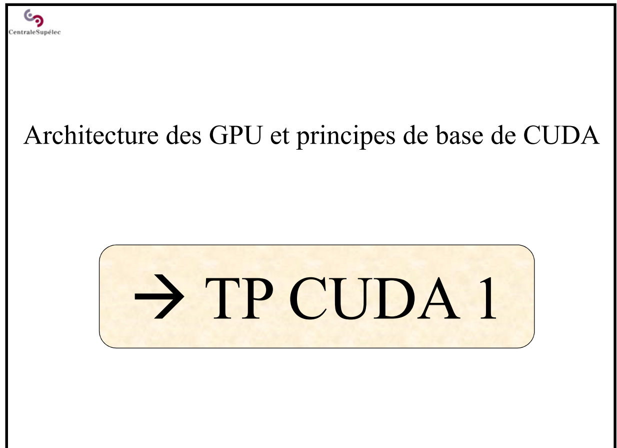
→ Le compilateur élimine les lignes de code inutiles.

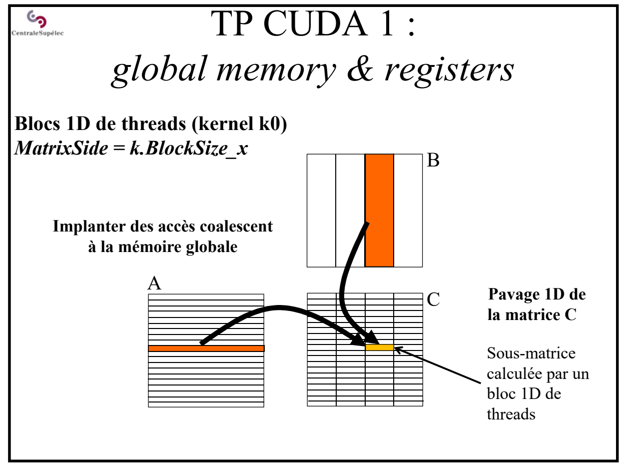
Exécution du kernel : `fk<SIZE><<<Dg,Db>>>(…)`

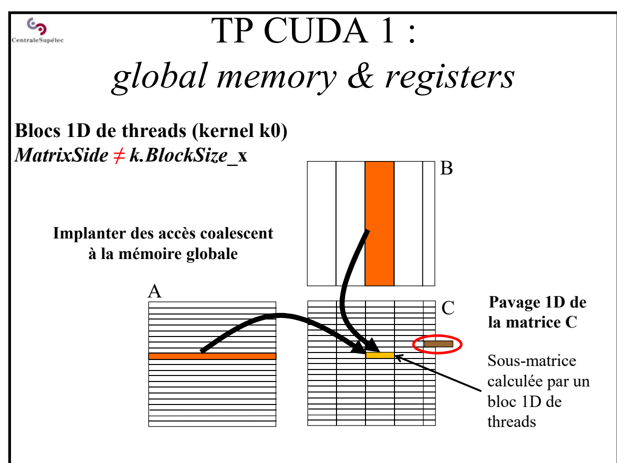
Permet de spécialiser à la compilation un kernel générique, et de gagner en efficacité

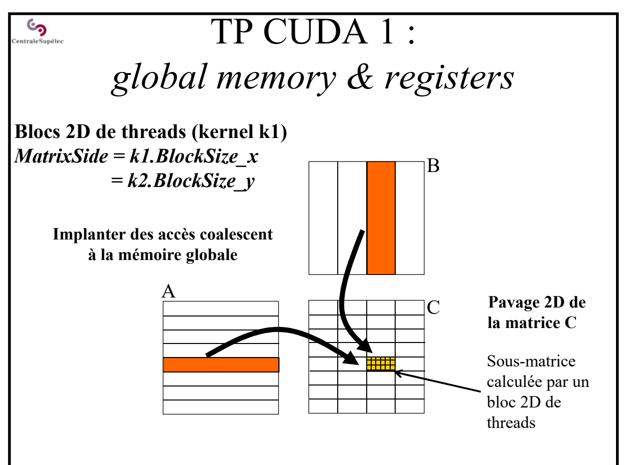

Démarche et contraintes de développement
Pb d'égalité des résultats CPU/GPU
Calculs en simple et double précision sur CPU et GPU :
• Les double sont disponibles et bien supportés
• La norme IEEE754 est respectée par les GPU
Mais selon l'ordre des opérations les résultats peuvent varier (classique)
→ Donc des calculs parallèles suivis de réductions peuvent différer sur GPU et sur CPU
…difficile de savoir si on est face à une erreur de développement ou à une simple conséquence du parallélisme !
• Les calculs en simple précision (float) mènent rapidement à des écarts selon l'ordre des opérations
→ Les calculs parallèles en float sur CPU et GPU divergent rapidement !

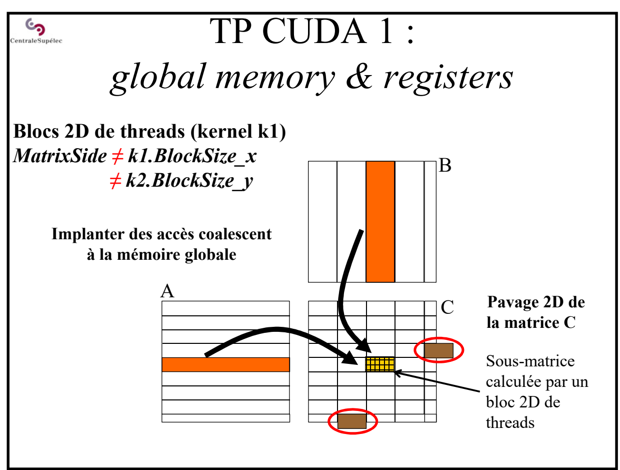

Démarche et contraintes de développement
Précision vs Vitesse des calculs
Calculs en float
• Si on veut se limiter à des float alors préciser aussi que les constantes sont des float : x = y*4.0f
Calculs en float avec précision limitée
• On peut positionner certains flags de compilation pour profiter de la vitesse de traitement des float avec une précision limitée
Compilation avec : --ftz=true --prec-sqrt=false
--prec-div=false --fmad=true
• On peut aussi forcer globalement l'utilisation de la bibliothèque « fast math » pour accélérer les calculs si la précision n'est pas critique :
Compilation avec --use_fast_math

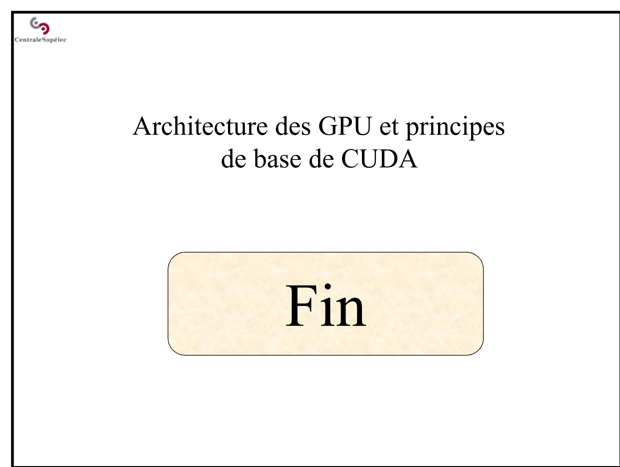

Architecture des GPU et principes de base de CUDA
→ TP CUDA 1


TP CUDA 1 :
global memory & registers
Blocs 1D de threads (kernel k0)
MatrixSide = k.BlockSize_x
Implanter des accès coalescent
à la mémoire globale
B
A
C
Pavage 1D de
la matrice C
Sous-matrice
calculée par un
bloc 1D de
threads


TP CUDA 1 :
global memory & registers
Blocs 1D de threads (kernel k0)
MatrixSide ≠ k.BlockSize_x
Implanter des accès coalescent
à la mémoire globale
B
A
C
Pavage 1D de
la matrice C
Sous-matrice
calculée par un
bloc 1D de
threads


TP CUDA 1 :
global memory & registers
Blocs 2D de threads (kernel k1)
MatrixSide = k1.BlockSize_x
= k2.BlockSize_y
Implanter des accès coalescent
à la mémoire globale
B
A
C
Pavage 2D de
la matrice C
Sous-matrice
calculée par un
bloc 2D de
threads


TP CUDA 1 :
global memory & registers
Blocs 2D de threads (kernel k1)
MatrixSide ≠ k1.BlockSize_x
≠ k2.BlockSize_y
Implanter des accès coalescent
à la mémoire globale
B
A
C
Pavage 2D de
la matrice C
Sous-matrice
calculée par un
bloc 2D de
threads


Architecture des GPU et principes
de base de CUDA
Fin