CentraleSupélec

Mineure HPC-SBD

Architecture des GPU et principes de base de CUDA

Stéphane Vialle

CentraleSupélec université PARIS-SACLAY — École doctorale Sciences et technologies de l'information et de la communication (STIC) — RISEGrid — Grand Est

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

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

CentraleSupélec

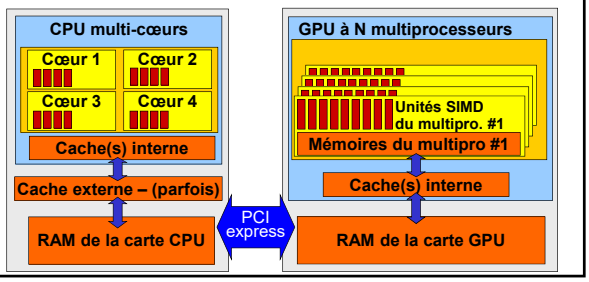
1 – Architecture d'un GPU NVIDIA

- 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 :

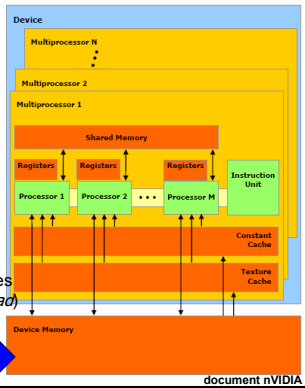


Architecture d'un GPU NVIDIA

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

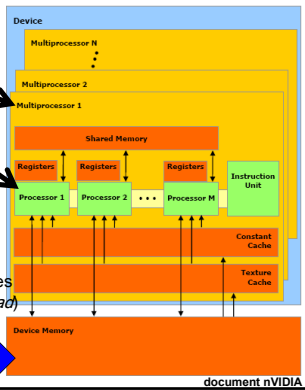
document nVIDIA

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 in

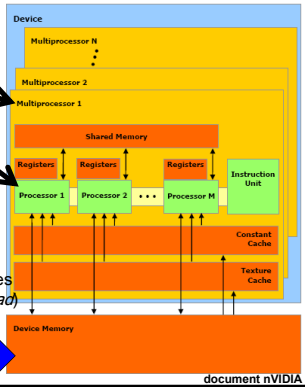
Terminologie 2 :

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

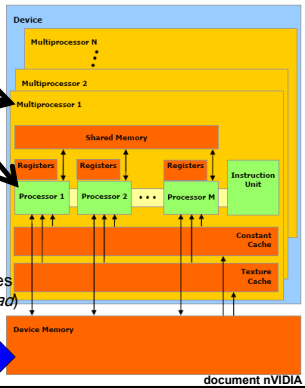
Terminologie « S. Vialle » :

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*)



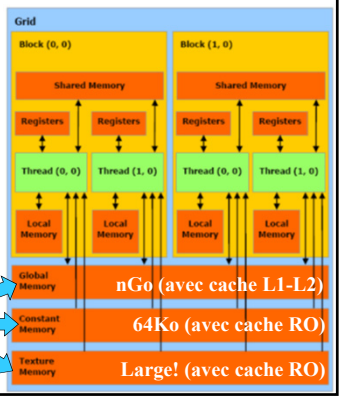
CPU + RAM

document nVIDIA

Architecture d'un GPU NVIDIA

Principaux concepts d'architecture

Une hiérarchie de mémoires de tailles et de temps d'accès très variables



CPU + RAM

Architecture d'un GPU NVIDIA

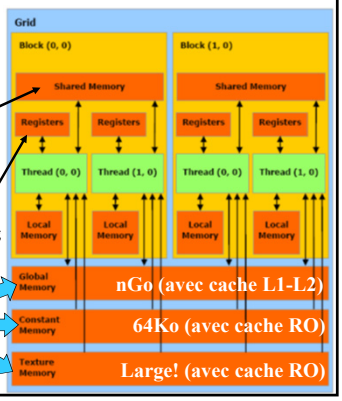
Principaux concepts d'architecture

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, **M**axwell, **P**ascal, **V**olta, **T**uring



Architecture d'un GPU NVIDIA

Principaux concepts d'architecture

Une hiérarchie de mémoires de tailles et de temps d'accès très variables

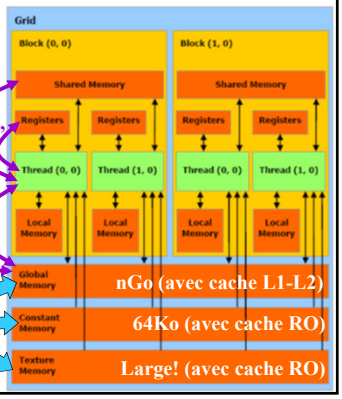
Comparé à la vitesse de calcul du GPU

Rapide et souple

'0s'

Moyen-lent & contraint

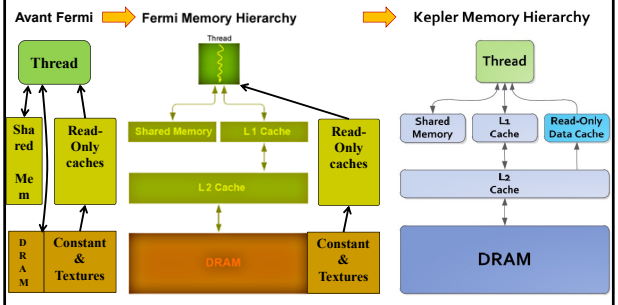
Très lent



Architecture d'un GPU NVIDIA

Cache génériques des GPU

Evolution des caches : de + en + de cache « générique »



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

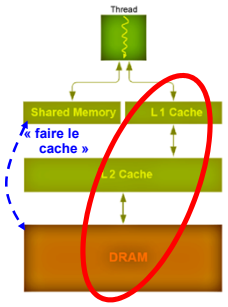
  ASSEZ SIMPLE
  LENT-RAPIDE

- 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*

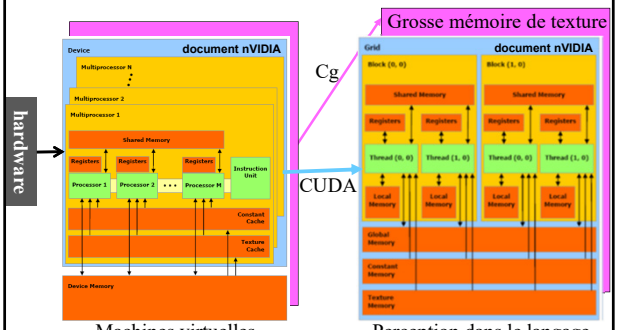
  TRES COMPLEXE
  LE PLUS RAPIDE *(ex : BLAS en CUDA)*



Architecture d'un GPU NVIDIA

Perception de l'architecture d'un GPU

La perception de l'architecture change selon le langage de prog.



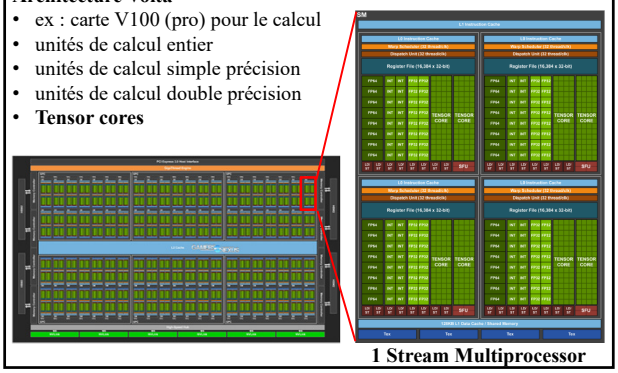
Machines virtuelles — Perception dans le langage

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**



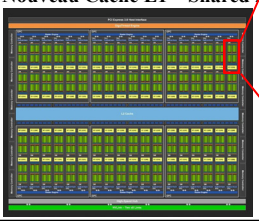
1 Stream Multiprocessor

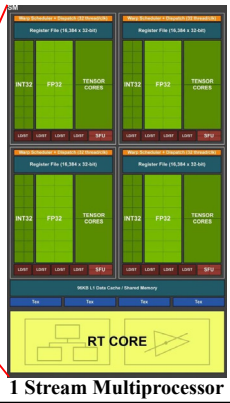
Architecture d'un GPU NVIDIA

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**





1 Stream Multiprocessor

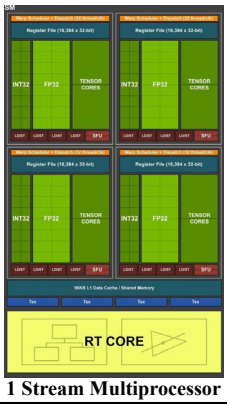
Architecture d'un GPU NVIDIA

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



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 !

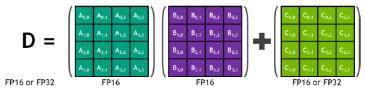
SOL MAN from NVIDIA SOL ray tracing demo running on a Turing TU102 GPU with NVIDIA RTX technology in real-time

Architecture d'un GPU NVIDIA

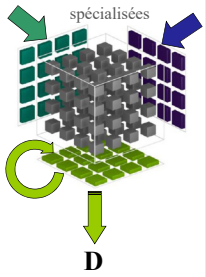
Nouveautés architecturales

Tensor cores

- **1** TC : Produit-Addition matricielle 4x4
  D = A.B + C, avec accumulation de produits



64 unités de calculs spécialisées



- Précision mixte :
  - flux d'opérandes sur 16 bits
  - calculs internes sur 32 bits
  - flux de résultats en 16 ou 32 bits

Architecture d'un GPU NVIDIA

Nouveautés architecturales

Tensor cores

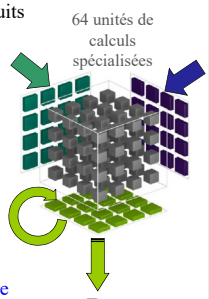
- **1** TC : Produit-Addition matricielle 4x4
  D = A.B + C, avec accumulation de produits
- Très nombreuses applications
  - Machine Learning
  - algèbre linéaire
  - traitement d'images…

Un Tensor core :

- est une implantation hardware d'un opérateur mathématique très utile dans les applications d'aujourd'hui,
- utile également aux calculs graphiques

→ Un opérateur qui mérité qu'on lui consacre une partie de la puce !

64 unités de calculs spécialisées



Architecture d'un GPU NVIDIA

Nouveautés architecturales

Cache L1 – Shared memory

- 96 Ko par SM
- Cache L1, shared memory, cache de texture : unifiés
- Nouvelle stratégie de gestion de cache (plus efficace)
- Si pas d'utilisation de la *shared memory* : tout pour le cache L1

  La « *shared memory* » est une mémoire cache L1 dépourvue d'algo de cache, que l'utilisateur doit implanter lui-même !

Objectifs des améliorations de l'architecture TURING :

- Augmenter les perfs des algo. pas adaptés à la *shared memory*
- Diminuer les pertes de perfs si on se contente du cache L1…

En fait … bcp d'utilisateurs refusent d'écrire un algo de cache !

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

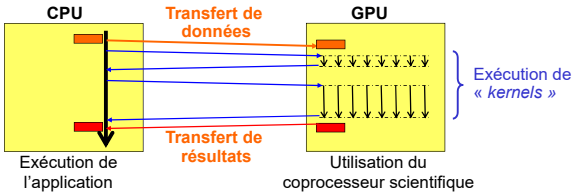
2 – Exécution d'un pgm CUDA

- Principe d'exécution
- Exec. de grilles de blocs de threads
- Exec. de blocs de threads par *warps*
- Granularité de la grille et des blocs

Principe d'exécution

Exécution d'applications CUDA :

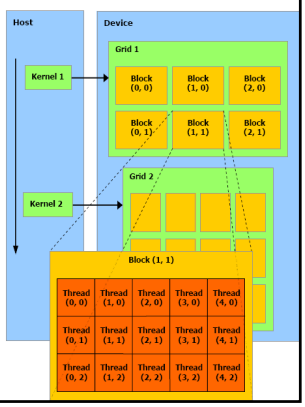
- On lance un programme CPU d'apparence classique.
  On réalise du « RPC » sur le GPU depuis le CPU : exécution de « kernels ».
- Il faut minimiser les transferts de données (pour être efficace).
- On peut exécuter les « kernels » en mode bloquant (synchrone) ou non-bloquant (asynchrone) vis-à-vis du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



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.

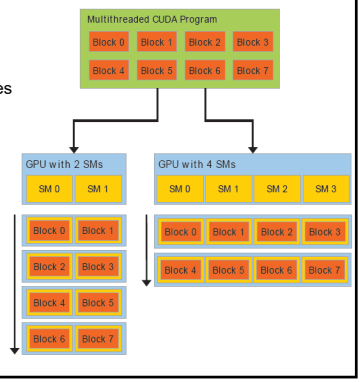


Exécution d'un programme CUDA

Exec. de grilles de blocs de *threads*

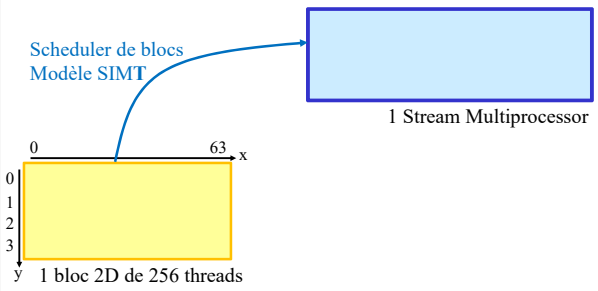
Le programme CPU demande l'exécution d'un ensemble de threads (des « gpu threads ») **:**

- le *scheduler* de blocs répartit les blocs sur les différents Multi-Processeurs
- différents GPU arriveront sans problème à exécuter la même grille de blocs



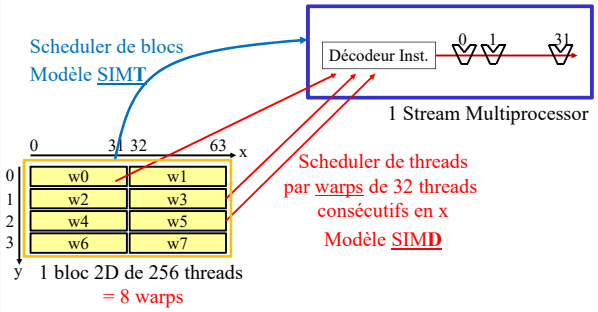
Exécution d'un programme CUDA

Exec. de blocs de threads par *warps*



Exécution d'un programme CUDA

Exec. de blocs de threads par *warps*



Exécution d'un programme CUDA

Granularité de la grille et des blocs

Créer des blocs contenant un nombre entier de warps

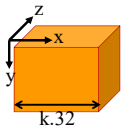
- un décodeur d'instruction pilote 32 threads hardware / ALU
- le scheduler de threads active des warps de 32 threads si possible consécutifs en x dans le bloc (il privilégie la dimension en x)

→ Créer des blocs **d'au moins 32 threads** sinon une partie des ALU seront toujours inutilisées (voir TP) !

→ Créer des blocs ayant **un nombre de threads multiple de 32** sinon le dernier warp sera incomplet et des ALU seront inutilisées

→ Créer des blocs ayant **une dimension en x multiple de 32** sinon la « coalescence » sera moins bonne ou plus compliquée. Voir plus loin….

Mais souvent ca marche encore très bien avec une dimension en x multiple de 16 (voir TP) !



k.32

Exécution d'un programme CUDA

Granularité de la grille et des blocs

Masquage des temps d'accès mémoires des GPU :

- un GPU passe d'un warp de threads à un autre très rapidement
- un GPU masque la latence de ses accès mémoires par multi-threading

→ Ne pas hésiter à créer un **grand nombre de petits threads** GPU par bloc et un grand nombre de blocs.

→ Pour traiter une table de N éléments :

- Threads traitant UN élément chacun
- Grille de blocs de N threads au total

vs

- Threads traitant n éléments chacun
- Grille de blocs de N/n threads au total

Exécution d'un programme CUDA

Granularité de la grille et des blocs

Combien de threads/bloc et de blocs/grille ?

Le *scheduler* des threads d'un blocs souhaite avoir « beaucoup de threads dans un bloc »

→ Avoir des warps de threads en réserve pour recouvrir des temps d'accès à la mémoire

Le *scheduler* de blocs souhaite avoir « plein de blocs pas trop gros »

→ Avoir des blocs en réserve pour utiliser tous les SM du GPU
→ Participer au multithreading et au recouvrement des temps d'accès RAM (chargement de plusieurs blocs « résidents » dans un SM)

Rmq : le GPU ne démarre un bloc de threads sur un multiprocesseur que s'il y a assez de registres et autres rsrc disponibles

→ faire des blocs pas trop gros permet d'en charger plus en résidents dans un multiprocesseur

Vaut-il mieux faire peu de gros blocs ou beaucoup de petits blocs ?

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**

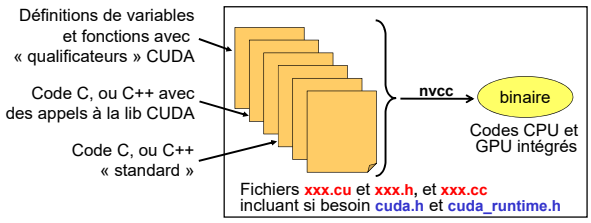
Architecture des GPU et principes de base de CUDA

3 – Compilation d'un pgm CUDA

- Compilation 100% CUDA
- Compilation mixte CUDA & C++

Compilation 100% CUDA

Compilation d'applications CUDA – entièrement développées en CUDA :



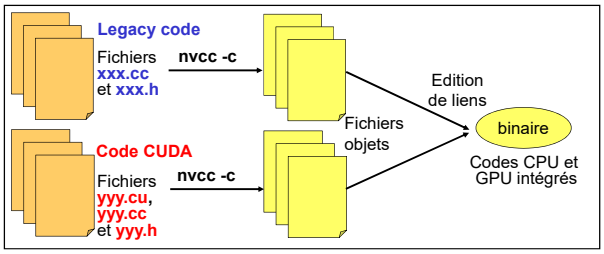
Pour les codes C/C++ simples :

→ Il est possible de simplement tout recompiler en **nvcc** dans des fichiers xxx.cu (et xxx.cc incluant *cuda.h* et *cuda_runtime.h*)

→ Mais les optimisations sérielles peuvent en souffrir…

Compilation mixte CUDA & C++

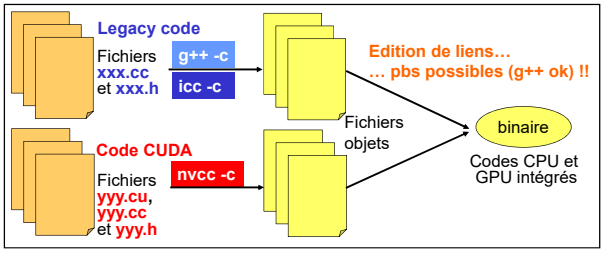
Compilation d'applications CUDA – avec récupération de code C/C++ :



- 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**

Architecture des GPU et principes de base de CUDA

4 – Programmation en CUDA à base de registres

- *Qualifiers* de CUDA
- Transfert de données CPU-GPU
- Variables statiques ou dynamiques ?
- Définition de la *grille* de *blocs*
- Exécution de la grille de blocs
- 1er *Kernel* (traitant 1 donnée par *thread*)

« 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	Mémoire constante GPU	Mémoire partagée d'un multiprocesseur
	Durée de vie de l'application	Durée de vie de l'application	Durée de vie du *block de threads*
	Accessible par les codes GPU et CPU	Ecrit par code CPU, lu par code GPU	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);
```

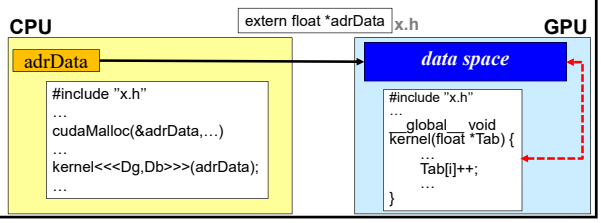
Libération des allocations dynamiques

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

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)



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 »

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

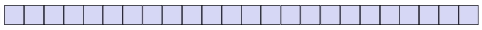
Définition de la grille :

- Barres, matrices, ou cubes de blocs.
- `Dg.x` ≤ $2^{31} - 1$
- `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
```

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

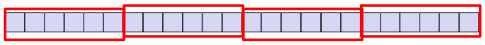
Définition des blocs (1D) :

Db.x = BLOCK_SIZE_X; // = 32/64/128/256/512/1024
Db.y = BLOCK_SIZE_Y; // = 1
Db.z = BLOCK_SIZE_Z; // = 1

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.

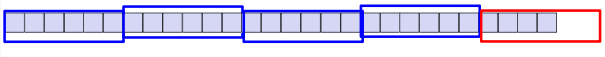
Exemple :
```
Dg.x = N/Db.x = N/BLOCK_SIZE_X
Dg.y = 1
Dg.z = 1
```

Et si (N % Db.x) ≠ 0 ?? … 

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)…

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;
```

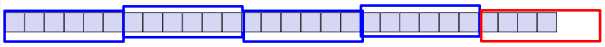
Rmq : on va créer « trop de threads » : il faudra en tenir compte dans le code (voir plus loin)…

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)

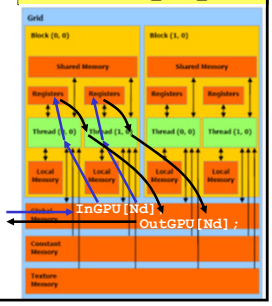
1er *Kernel* (traitant 1 donnée par *thread*)

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

```
__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;
}
```

```
Db = {BLOCK_SIZE_X,1,1}
Dg = {Nd/BLOCK_SIZE_X,1,1}
```

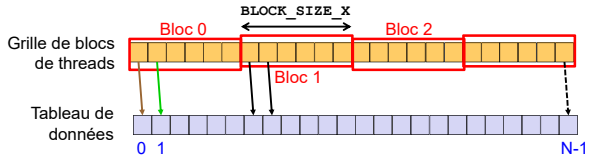


1er *Kernel* (traitant 1 donnée par *thread*)

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`



`idx = blockIdx.x*BLOCK_SIZE_X + threadIdx.x`

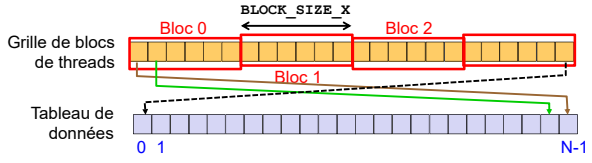
Indexage permettant des accès *coalescents*

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`



`idx = (N-1) - (blockIdx.x*BLOCK_SIZE_X + threadIdx.x)`

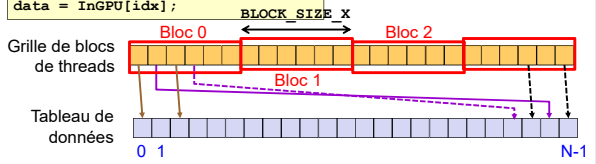
Mais ce serait un indexage *moins coalescent*… !!

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 = threadIdx.x +
      blockIdx.x*BLOCK_SIZE_X;
if (idx % 2 == 1)
   idx = (N-1) - idx;
// Read data from the global mem
data = InGPU[idx];
```

2 variables implicites et propres à chaque thread :
- dim3 threadIdx
- dim3 blockIdx




```
idx = (N-1) - (blockIdx.x*BLOCK_SIZE_X + threadIdx.x)
```

Mais ce serait un indexage *NON coalescent*… !!

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

```
__global__ void k1(void)
{
  int idx;        // Registers:
  float data;     // 16Kreg per
  float res;      // multipro.
  // Compute data idx of the thread
  idx = threadIdx.x +
        blockIdx.x*BLOCK_SIZE_X;
  // If the elt indexed exists:
  if (idx < Nd) {
    // 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;
  }
}
```

```
Db = {BLOCK_SIZE_X,1,1}
if (Nd%BLOCK_SIZE_X == 0)
  Dg = {Nd/BLOCK_SIZE_X,1,1}
else
  Dg = {Nd/BLOCK_SIZE_X + 1,1,1}
```

Démarche classique :

Les threads « en trop » ne font rien…

Architecture des GPU et principes de base de CUDA

5 – Respect de la « coalescence »

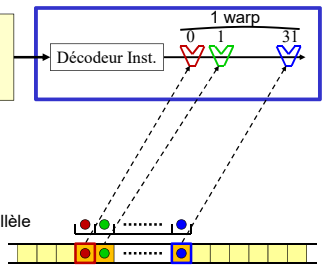
- *Thread* lisant 1 donnée sur tableau 1D
- *Thread* lisant 1 colonne sur tableau 2D
- *Thread* lisant 1 ligne sur tableau 2D
- *Thread* lisant *n* données sur tableau 1D
- Sensibilité de la coalescence
- Règles de développement

Respect de la coalescence

Thread lisant 1 donnée sur tableau 1D

Exemple 1D coalescent :

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
float data = Tab[idx];
....
```



32 data accédées en parallèle
(1 chargeur)

Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

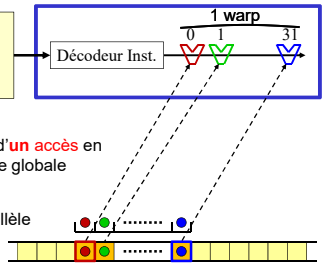
@ k.32.4Bytes @ (k.32+31).4Bytes

Respect de la coalescence

Thread lisant 1 donnée sur tableau 1D

Exemple 1D coalescent :

```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
float data = Tab[idx];
....
```



Coalescence parfaite !

Temps d'un accès en mémoire globale

32 data accédées en parallèle
(1 chargeur)

Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

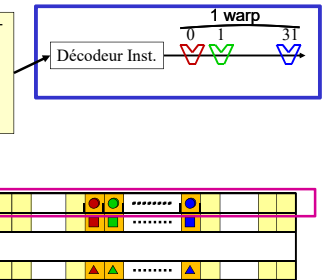
@ k.32.4Bytes @ (k.32+31).4Bytes

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

n = 0 :

Conditions :

- Accès mémoire contigus
- Démarrage à une adresse multiple de 32 mots mémoire

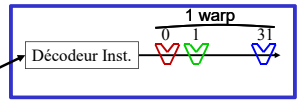
@ k.32.4Bytes @ (k.32+31).4Bytes

Respect de la coalescence

Thread lisant 1 colonne sur tableau 2D

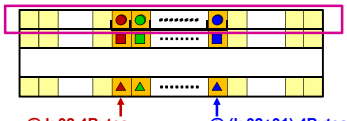
Exemple 2D coalescent :

```
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

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

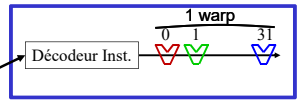
Respect de la coalescence

Thread lisant 1 colonne sur tableau 2D

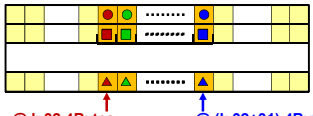
Exemple 2D coalescent :

```
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



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

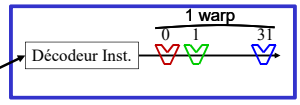
Respect de la coalescence

Thread lisant 1 colonne sur tableau 2D

Exemple 2D coalescent :

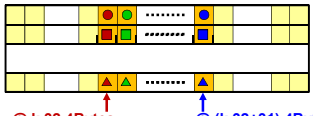
```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[n][idx];
  ....
}
```

Chaque thread lit une colonne



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

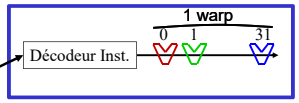
@ k.32.4Bytes @ (k.32+31).4Bytes

Respect de la coalescence

Thread lisant 1 colonne sur tableau 2D

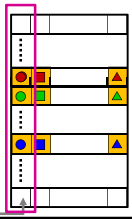
Exemple 2D non coalescent :

```
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

n = 0 : ....



Hypothèse (peu importante) :
- Démarrage à une adresse multiple de 32 mots mémoire

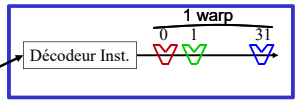
@ k.32.4Bytes

Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

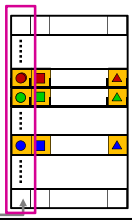
Exemple 2D non coalescent :

```
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

n = 0 : ....



Hypothèse (peu importante) :
- Démarrage à une adresse multiple de 32 mots mémoire

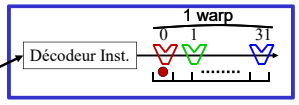
@ k.32.4Bytes

Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

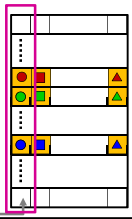
Exemple 2D non coalescent :

```
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

n = 0 : ....



Hypothèse (peu importante) :
- Démarrage à une adresse multiple de 32 mots mémoire

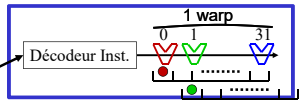
@ k.32.4Bytes

Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

Exemple 2D non coalescent :

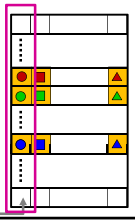
```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[idx][n];
  ....
}
```



Chaque thread lit une ligne

n = 0 : ....

Hypothèse (peu importante) :
- Démarrage à une adresse multiple de 32 mots mémoire

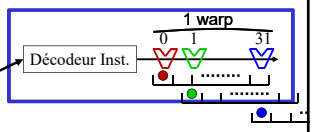


Respect de la coalescence

Thread lisant 1 ligne sur tableau 2D

Exemple 2D non coalescent :

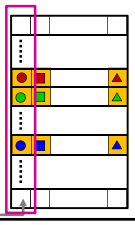
```
int idx = blockIdx.x*BLOCK_SIZE_X +
          threadIdx.x;
for (int n = 0; n < N; n++) {
  float data = Tab[idx][n];
  ....
}
```



Chaque thread lit une ligne

n = 0 : 32 accès en séquentiel !!

Hypothèse (peu importante) :
- Démarrage à une adresse multiple de 32 mots mémoire



Aucune coalescence !

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}
```

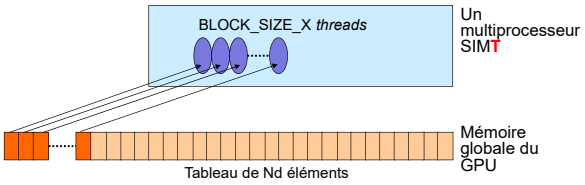
```
__global__ void f1(void)
{
  int offset = 0;                 // Registers
  float data = 0.0f, res = 0.0f;
  // Compute intitial data idx of the thread
  offset = threadIdx.x + blockIdx.x*(BLOCK_SIZE_X*n);
  // Loop with contiguous accesses to data tables
  for(int i = offset;
      i < offset + 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 :

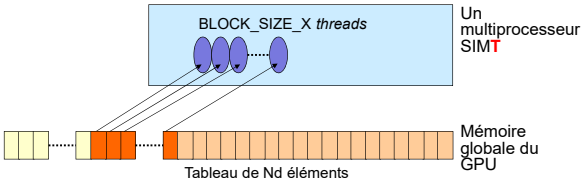


Step 1:
BLOCK_SIZE_X accès contigus, et synchronisés par warp
1 warp : modèle *Single Instruction Multiple Data*
1 bloc : modèle SIMT

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :

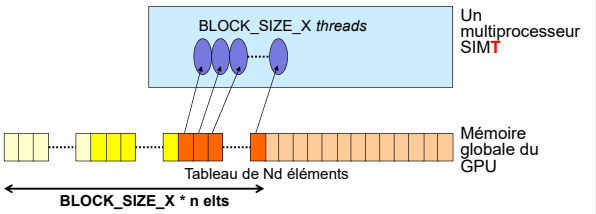


Step 2:
BLOCK_SIZE_X accès contigus, et synchronisés par warp
1 warp : modèle *Single Instruction Multiple Data*
1 bloc : modèle SIMT

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :

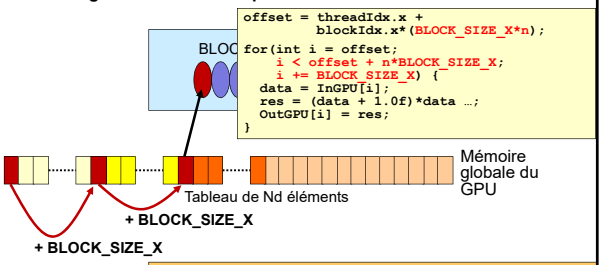


Step 3:
BLOCK_SIZE_X accès contigus, synchro par warp
1 warp : modèle *Single Instruction Multiple Data*
1 bloc : modèle SIMT

Respect de la coalescence

Thread lisant *n* données sur tableau 1D

Accès contigus aux données depuis un bloc 1D de threads :



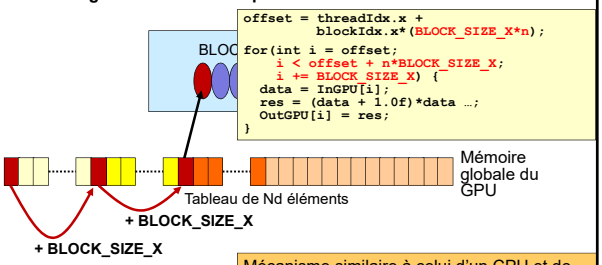

```
offset = threadIdx.x +
         blockIdx.x*(BLOCK_SIZE_X*n);
for(int i = offset;
    i < offset + n*BLOCK_SIZE_X;
    i += BLOCK_SIZE_X) {
  data = InGPU[i];
  res = (data + 1.0f)*data …;
  OutGPU[i] = res;
}
```

→ 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;
}
```

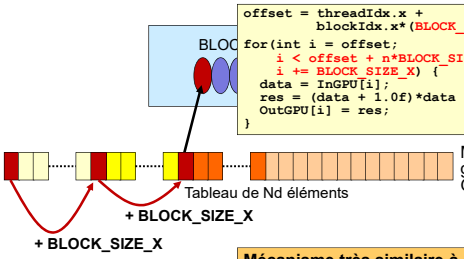
Mécanisme similaire à celui d'un CPU et de ses accès mémoires à travers son cache…
…**mais une boucle invisible est faite par des threads synchronisés**.

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;
}
```



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;
  }
}
```

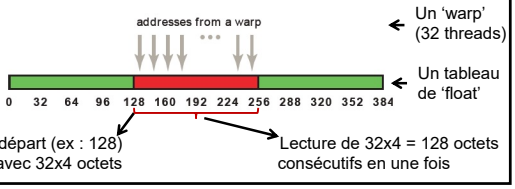
Respect de la coalescence

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



Respect de la coalescence

Sensibilité de la coalescence

Impact du « désalignement » :

- Hypothèse : les 32 threads d'un warp accèdent à 32 données consécutive, MAIS la première adresse n'est pas un multiple de 32x4o = 128o….
- Alors il y aura lecture de **DEUX** segments de 128 octets (au lieu d'un)

← Un 'warp' (32 threads)

← Un tableau de 'float'

Accès à 2 segments de 128 octets
→ 2 temps d'accès

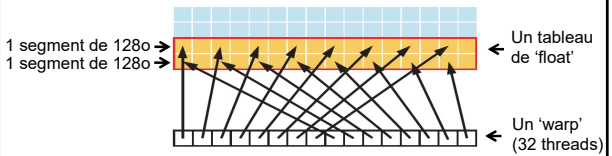
Respect de la coalescence

Sensibilité de la coalescence

Impact du « stride » (> 1) :

- Stride de 2 : le thread i accède à Tab[offset+ (2*i)]

  Rmq : on suppose le point de départ aligné (offset = k x 128)



← Un tableau de 'float'

← Un 'warp' (32 threads)

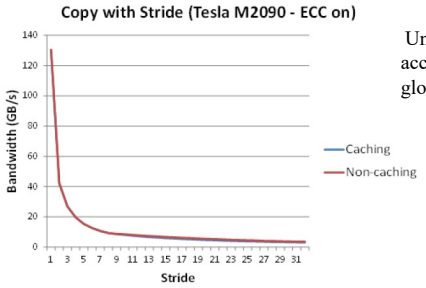
- On accède à 2 segments de 128 octets
  → 2 temps d'accès

Respect de la coalescence

Sensibilité de la coalescence

Impact du « stride » (> 1) :

- Dégradation de bande-passante applicative observée par NVIDIA



Un *stride* dans les accès à la mémoire globale se paie très vite très fort

Respect de la coalescence

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.

Respect de la coalescence

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

Limitation de la divergence

Exécution d'un « if…then…else »

Les divergences sont sources de ralentissement sur les archi. SIMD :

Ex : `if (x < 10) then {….} else {…}`

Exécution dans un warp :

1. Tous les threads testent la condition `(x < 10)`
2. Tous les threads qui doivent exécuter le *then* le font en parallèle
3. Tous les threads qui doivent exécuter le *else* le font en parallèle

Temps d'exécution :

- Si tous les threads exécutent le *then* :
  T(condition) + T(then)
- Si tous les threads exécutent le *else* :
  T(condition) + T(else)
- Sinon au moins un *then* et au moins un *else* sont exécutés :
  T(condition) + T(then) + T(else)

Limitation de la divergence

Exécution d'un « if…then…else »

Divergence très couteuse :

```
if (threadIdx.x % 2 == 0)
  {…}
else
  {…}
```

Chaque warp exécutera le *then* puis le *else*

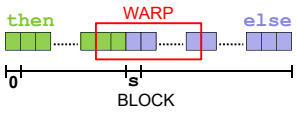
→ exécution lente sur tout le bloc !

Divergence moins couteuse :

```
if (threadIdx.x < s)
  {…}
else
  {…}
```

→ exécution lente dans un seul warp

En 1D un seul warp exécutera le *then* puis le *else* :



Limitation de la divergence

Exécution d'un « if…then…else »

Divergence très couteuse :

```
if (threadIdx.x % 2 == 0)
  {…}
else
  {…}
```

Chaque warp exécutera le *then* puis le *else*

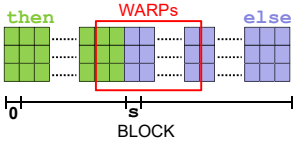
→ exécution lente sur tout le bloc !

Divergence moins couteuse :

```
if (threadIdx.x < s)
  {…}
else
  {…}
```

→ exécution lente dans une seule colonne de warps

En 2D une colonne de warps exécutera le *then* puis le *else* :



Architecture des GPU et principes de base de CUDA

7 – Démarche de développement

- Développer *Step-by-Step*
- Optimisation par *templates C++*
- Pb d'égalité des résultats CPU/GPU
- Précision *vs* Vitesse des calculs

Développement pas à pas

Développer et tester *step by step* !

- Certaines erreurs ne sont pas détectées à la compilation.
- Certains messages d'erreurs à l'exécution sont approximatifs.
- Certaines erreurs d'exécution sont « fatales » au driver et peuvent mener à rebooter le PC (très rare)
- Il existe des debugers sous Windows, mais le debug sur GPU reste complexe
- Certaines erreurs ne provoque pas d'arrêt du programme, seulement des résultats « légèrement » faux !

→ Développer, tester et valider successivement chaque kernel.

→ Comparer les résultats sur GPU et sur CPU pour valider les kernels.

En général, on ne peut pas développer un code GPU sans disposer ou développer le code CPU correspondant (pour comparer les résultats)

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

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 !

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*4.0**f**

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`**

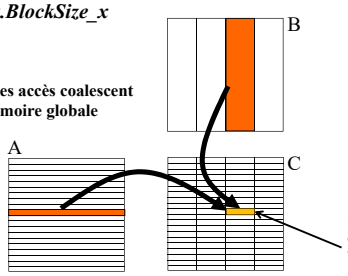
Architecture des GPU et principes de base de CUDA

→ 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



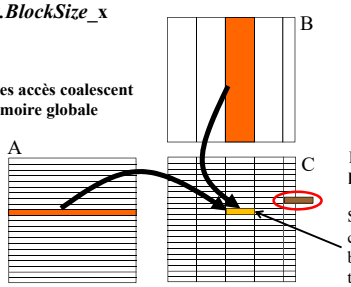
Pavage 1D de la matrice C

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



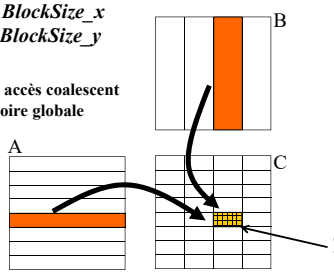
Pavage 1D de la matrice C

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



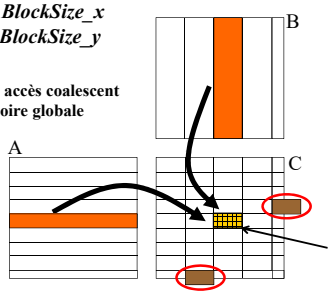
Pavage 2D de la matrice C

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



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