

Faculté
des Sciences
& Techniques



Université
de Limoges

Master 1^{ère} année

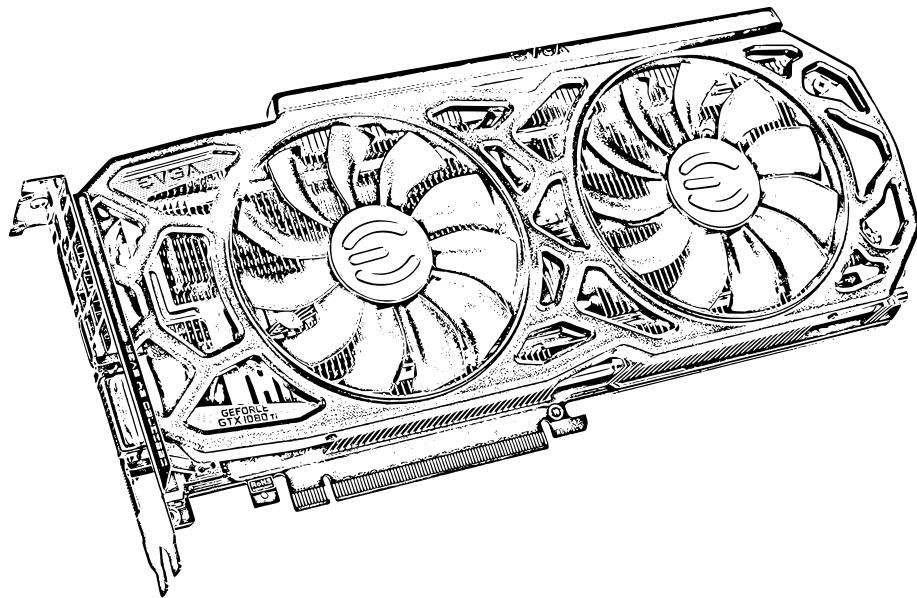


Table des matières

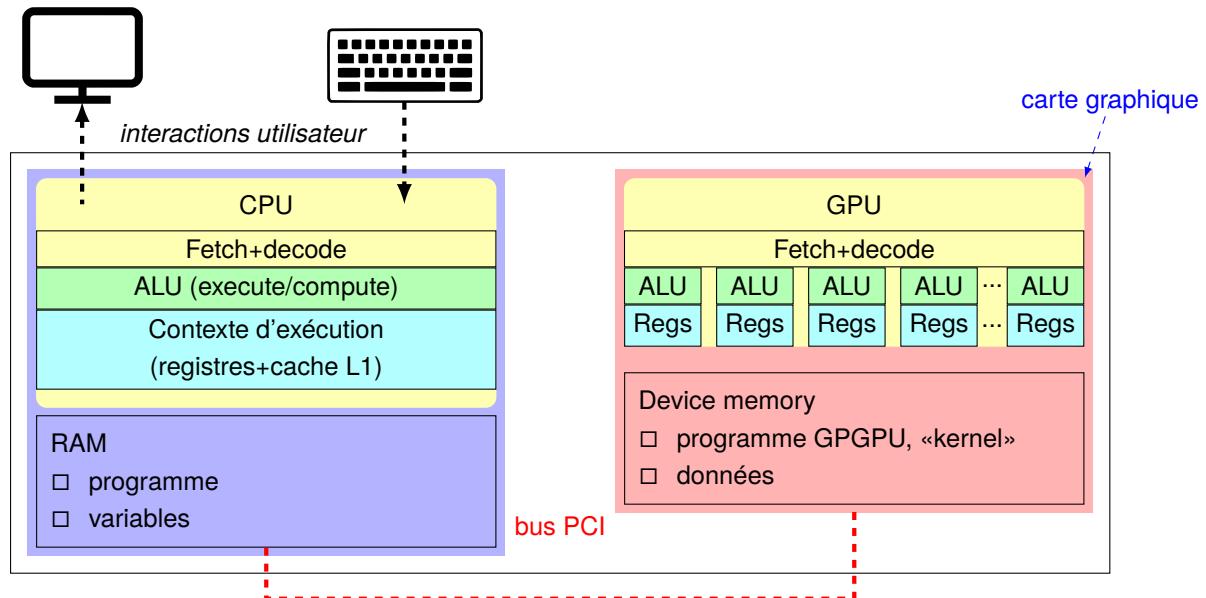
1	CUDA est un modèle SIMD	3
	L'architecture CUDA, « <i>Compute Unified Device Architecture</i> »	6
	Comment programmer ?	8
	Comment est gérer la mémoire ?	9
	Comment déclencher le travail sur le GPU ?	11
2	CUDA, « <i>Compute Unified Device Architecture</i> »	13
	La hiérarchie mémoire	15
	Répartition du travail entre threads regroupées en bloc	16
	Un seul programme source mixte CPU/GPU	18
	Communication entre «l'hôte» et le « <i>CUDA device</i> »	20
	Exécution d'une application parallèle sur le « <i>device</i> »	22
3	La notion de divergence	31
	Comparaison de performance divergence/pas de divergence	32
	Synchronisation & Communication	35
4	Aggrégation, « <i>coalescence</i> », des accès mémoire	38
	Optimisation de la vitesse en localisant mieux les données	43
	Stratégie de développement	44
	Extraction du parallélisme de données et la définition de «grille»	49
	Optimisation	51



1 CUDA est un modèle SIMD

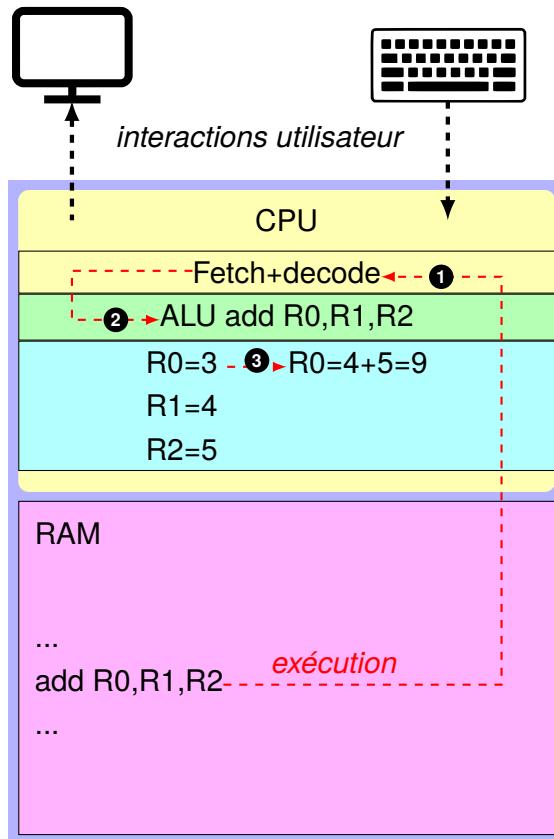
3

- Le CPU est composé de :
- une unité de contrôle chargée de :
 - ◊ chercher en mémoire les instructions à exécuter, «*fetch*» ;
 - ◊ décoder ces instructions en termes d'opération à faire sur les registres et la mémoire ;
 - une unité ALU, «*Arithmétique et Logique*» ;
 - des registres et de la mémoire cache pour limiter les accès à la RAM ;



- Le GPU est composé de :
- une unité de contrôle ;
 - nombreuses unités ALU+Registres combinées (plusieurs milliers).





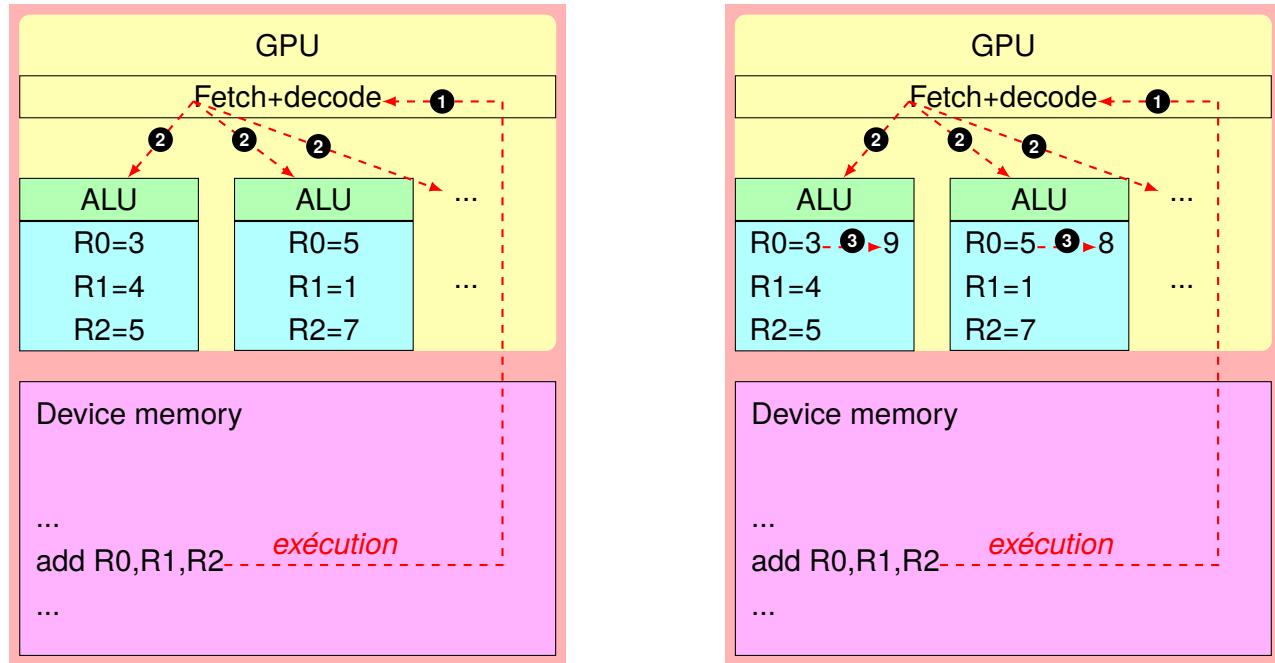
Déroulement de l'exécution d'une instruction sur le CPU :

- ① ⇒ une instruction est récupérée depuis la RAM et décodée ;
- ② ⇒ elle déclenche des opérations de l'ALU sur les registres et/ou le contenu de la mémoire ;
- ③ ⇒ le résultat est rangé dans un registre ;
- ④ ⇒ on recommence sur l'instruction suivante.



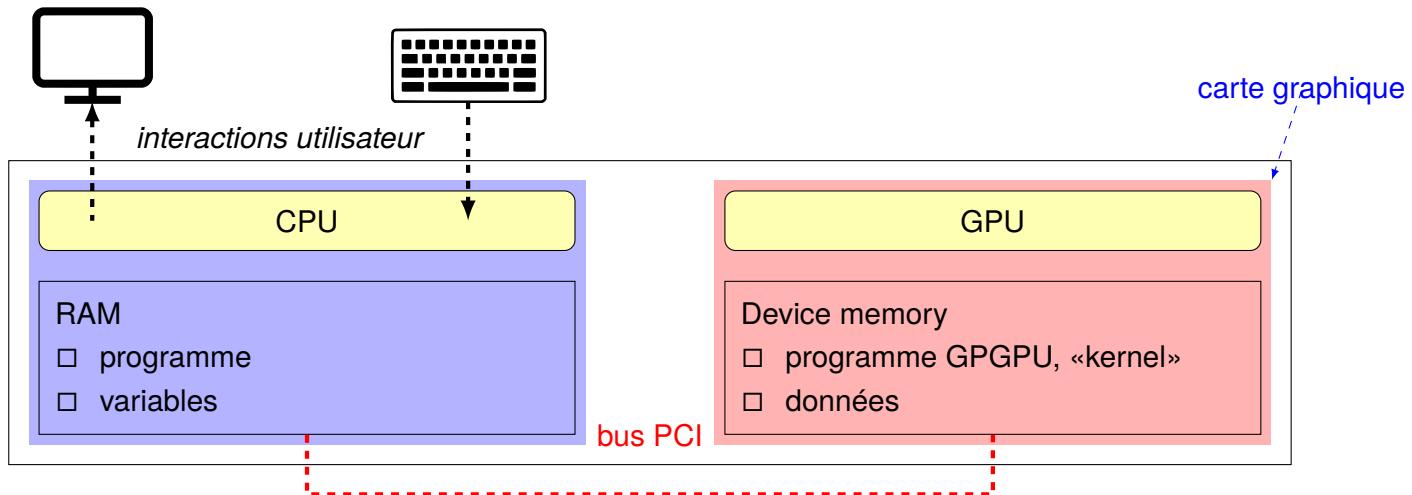
L'exécution sur le GPU

5



Déroulement de l'exécution d'une instruction sur le GPU :

- ①⇒ une instruction est récupérée depuis la RAM et décodée ;
- ②⇒ elle déclenche des opérations identiques sur les différentes ALU entre les registres associés et/ou le contenu de la mémoire ;
- ③⇒ le résultat est rangé dans un registre local.



Architecture CUDA : le système «host», CPU, et le système GPU, la carte graphique sont **séparés** :

▷ la RAM ou la mémoire de l'hôte est accessible uniquement par le CPU ;

▷ la «Device Memory» est accessible uniquement par le GPU ;

⇒ les **données** conservées dans la RAM ou la «*memory device*» doivent être **échangées** entre les deux à l'aide du bus PCI en mode DMA ;

⇒ un **programme** qui utilise le GPU est constitué de deux parties :

- ◊ une partie tournant sur le **CPU**, c-à-d sur l'hôte ;
- ◊ une partie tournant sur le **GPU**, c-à-d sur le «*device*».



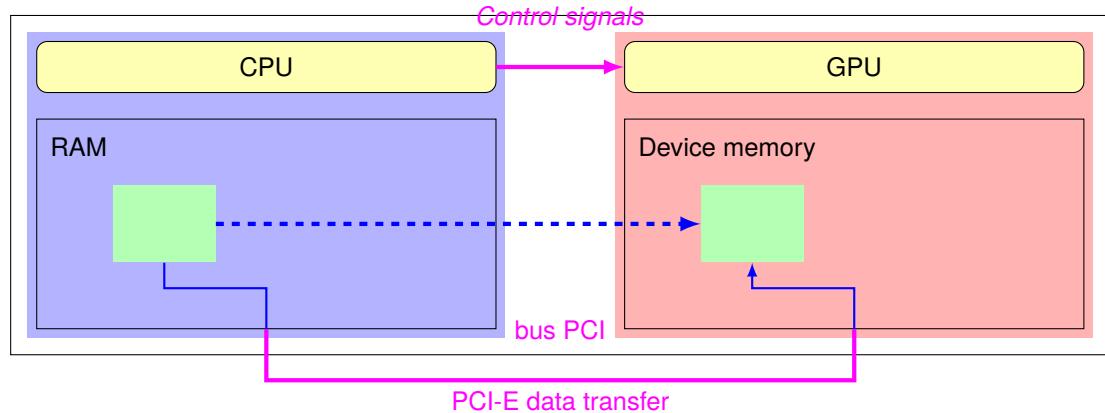
Comment faire travailler le GPU sur les données ?

7

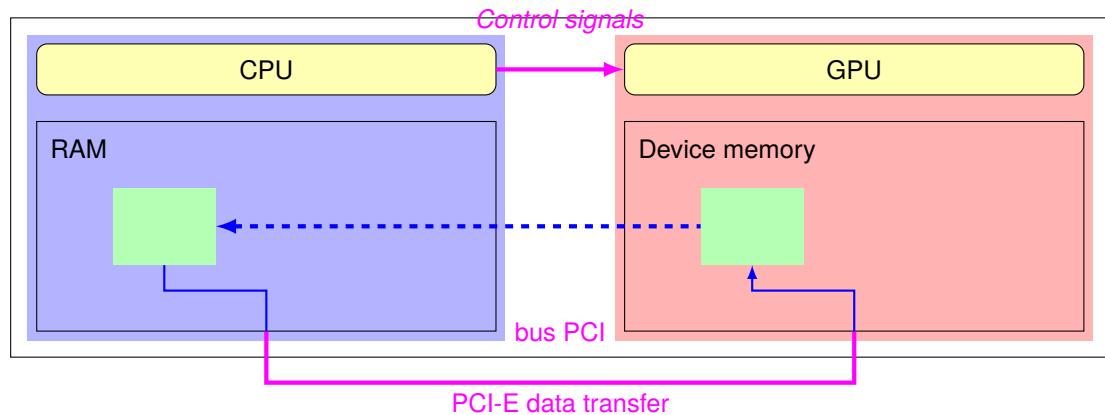
CUDA met à disposition des fonctions de transfert de données entre la mémoire RAM et la mémoire du «device» :

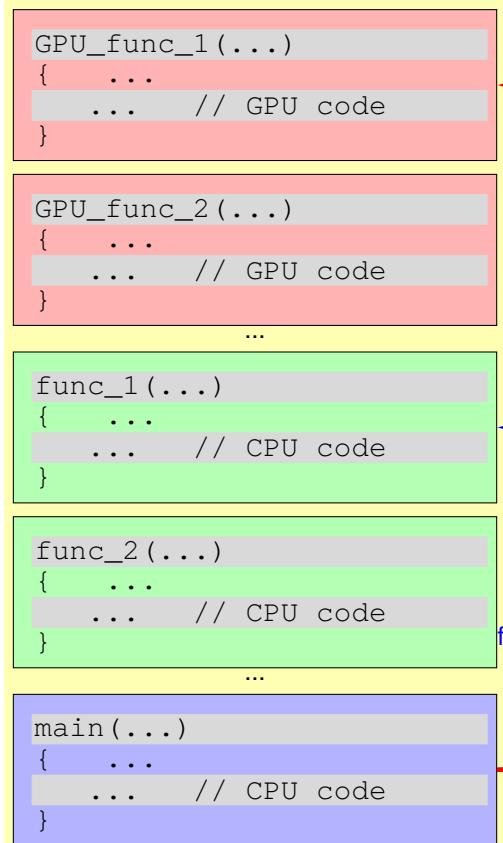
De la RAM vers la mémoire du «device» :

Des signaux de contrôle sont utilisés par le CPU pour déclencher l'opération sur le GPU.



De la mémoire du «device» vers la RAM :





⇒ une fonction GPU ne retourne pas de valeur !

Le programmeur écrit **un seul programme source** constitué de deux types de fonctions :

- ▷ les fonctions `func_x` sont exécutées par le CPU comme des fonctions ordinaires ;
- ▷ les fonctions `GPU_func_x` sont exécutées par le GPU sur la carte graphique ;
- ▷ la fonction principale, `main`, exécutée par le CPU appelle les différents types de fonctions : c'est elle qui est appelée en premier au lancement du programme.

Il existe **deux types de fonction** CUDA :

- ▷ précédées par `__global__` : peuvent être appelées par le «*host*» ou par une autre fonction du «*device*» ;


```
__global__ void GPU_function1 ( parameters) {...}
```
- ▷ précédées par `__device__` : ne peuvent être appelées que par une autre fonction du «*device*»


```
__device__ void GPU_function2 ( parameters) {...}
```

Les deux types de fonctions `__global__` et `__device__` ne doivent rien renvoyer (`void`).



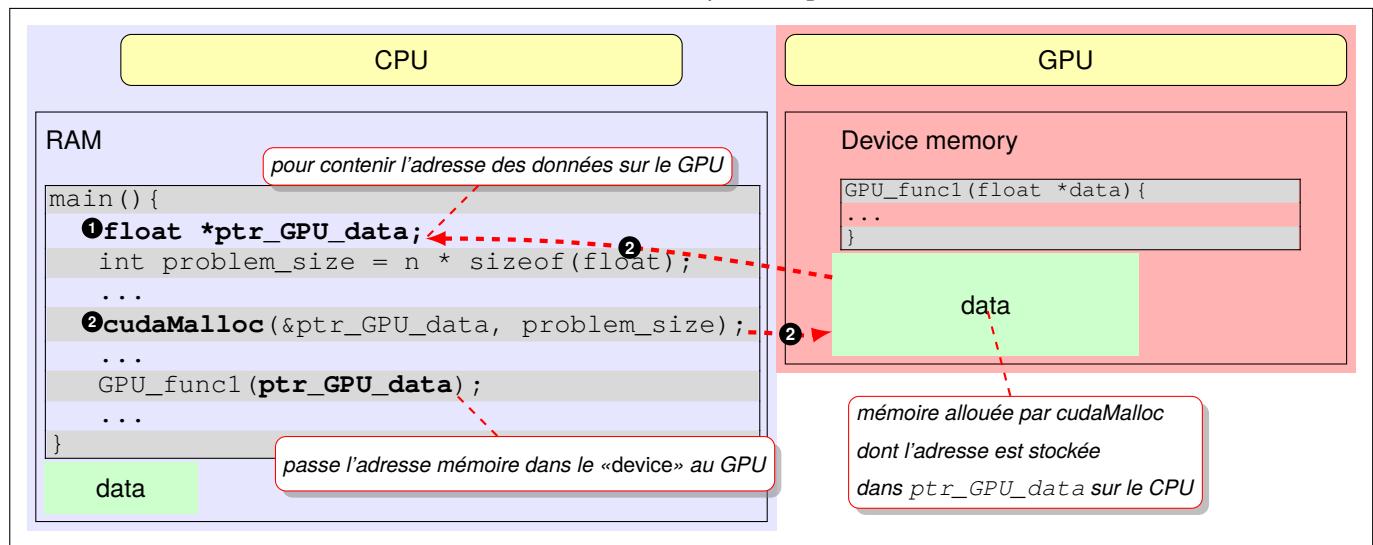
Comment est gérer la mémoire ?

9

Le «device» ne dispose pas d'OS, «Operating System» : il ne sait pas gérer sa mémoire !

⇒ C'est le «host» qui gère la mémoire pour le GPU :

- ▷ il déclare une variable du type pointeur sur le type de données à manipuler type *ptr ①;
- ▷ il alloue de la mémoire sur le GPU grâce à la fonction cudaMalloc (&ptr, nombre_octets) ②:
 - de la mémoire est «réservée» sur le GPU (c'est le CPU qui contrôle les espaces mémoires du device);
 - l'adresse de cette zone mémoire est stockée dans le pointeur ptr;



- ▷ lors de l'appel de la fonction GPU_func1, le CPU transmettra en paramètre l'adresse de la zone mémoire allouée précédemment stockée dans ptr à la fonction.

⇒ la fonction GPU GPU_func1 peut travailler sur la zone mémoire du device réservée à cet usage.

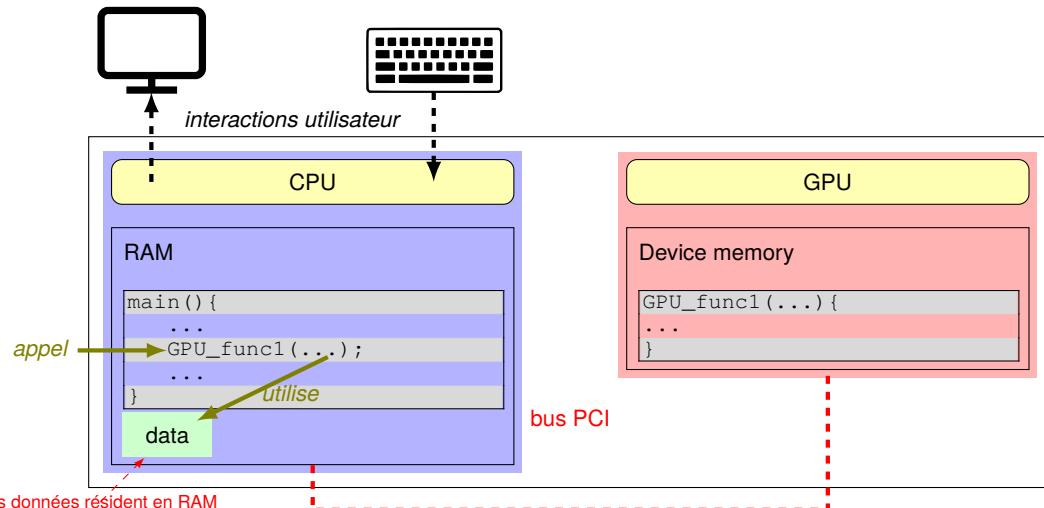


Comment déclencher le travail entre le CPU et le GPU ?

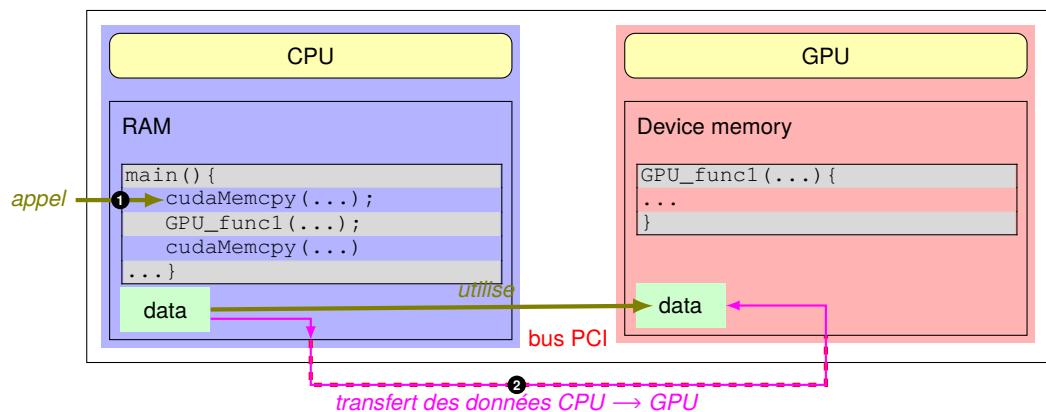
10

L'utilisateur n'interagit uniquement avec le programme CPU :
⇒ les données sont **unique-
ment** dans la RAM accessible par le CPU.

⇒ l'appel de la fonction GPU `GPU_func1()` ne peut être fait directement.

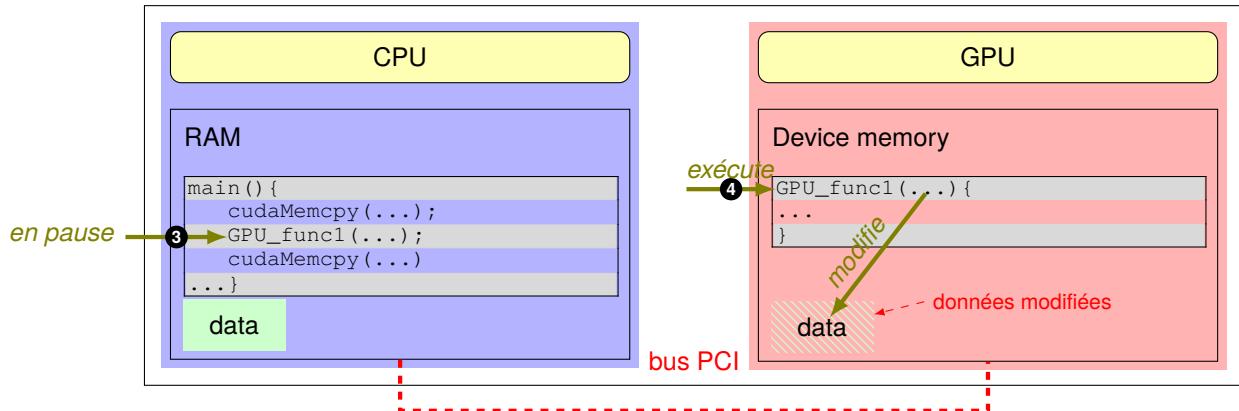
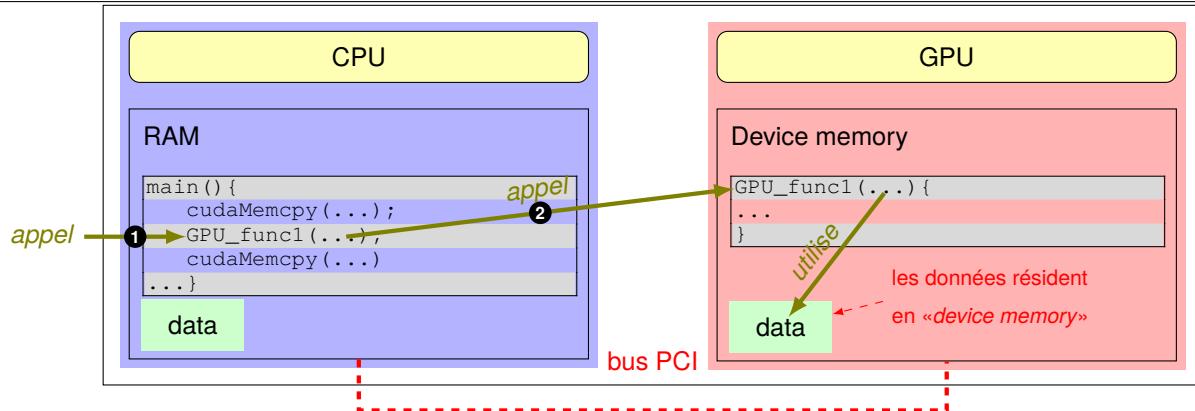


Les données sont transférées du CPU vers la mémoire «device» grâce à une opération `cudaMemcpy()` : les données sont transférées au travers du bus PCI.



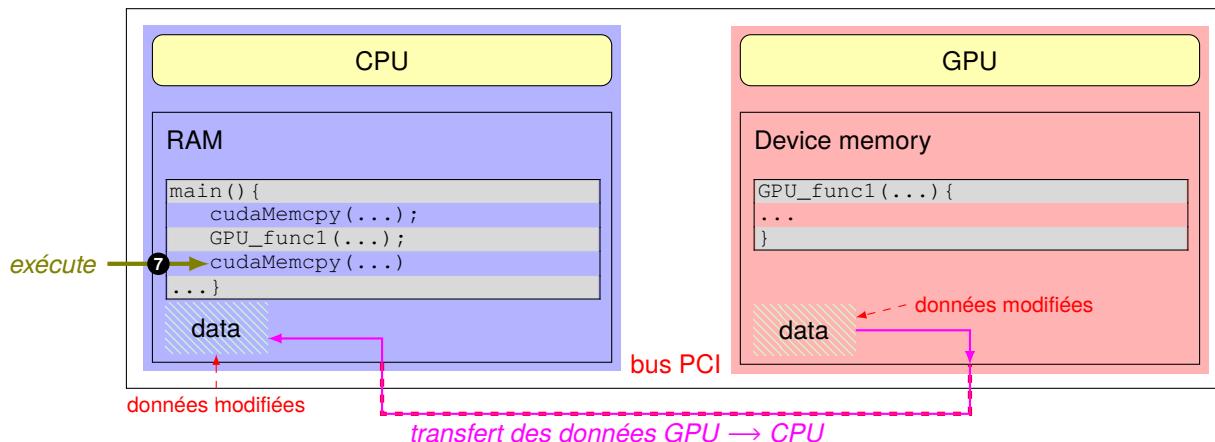
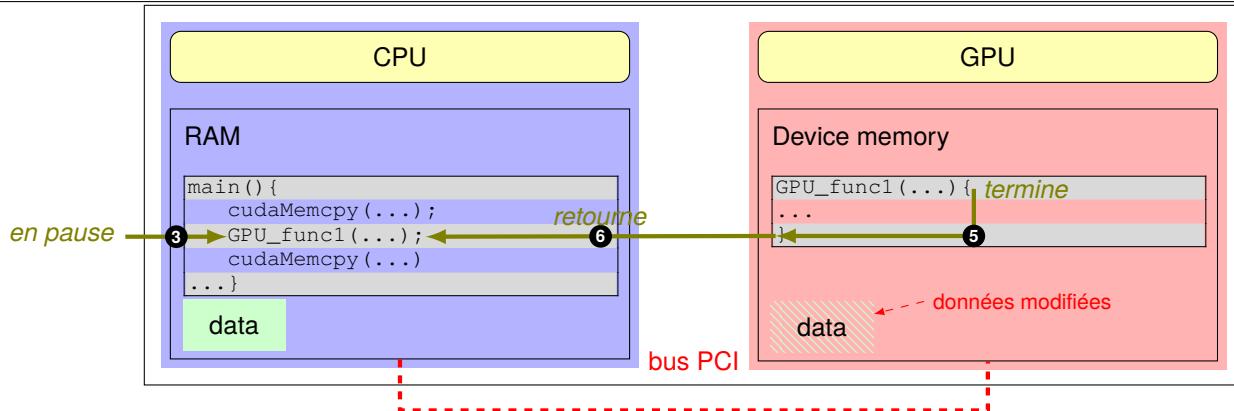
Comment déclencher le travail sur le GPU ?

11



Comment déclencher le travail sur le GPU ?

12



C'est un **environnement logiciel** qui permet d'utiliser le GPU, «*Graphics Processing Unit*» au travers de programme de haut niveau comme le C ou le C++ :

- ◊ le programmeur écrit un programme C avec des extensions CUDA, de la même manière qu'un programme OpenMP ;
- ◊ CUDA nécessite une carte graphique équipée d'un processeur NVIDIA de type Fermi, GeForce 8XXX/Tesla/Quadro, etc.
- ◊ les fichiers source doivent être compilés avec le compilateur C CUDA, NVCC.

Un **programme CUDA** utilise des «*kernels*» pour traiter des «*data streams*», ou «flux de données».

Ces flux de données peuvent être par exemple, des vecteurs de nombres flottants, ou des ensembles de frames pour du traitement vidéo.

Un «*kernel*» est exécuté dans le GPU en utilisant des threads exécutées en parallèles.

CUDA fournit **3 mécanismes** pour paralléliser un programme :

- ◊ un **regroupement hiérarchique** des threads ;
- ◊ des **mémoires partagées** ;
- ◊ des **barrières de synchronisation**.

*Ces mécanismes fournissent du parallélisme à **grain fin** imbriqué dans du parallélisme à **gros grain**.*



Des définitions

Terme	Définition
Host ou CPU	c'est l'ordinateur qui sert d'interface avec l'utilisateur et qui contrôle le «device» utilisé pour exécuter les parties de calcul intensif basé sur un parallélisme de données. <i>L'hôte est responsable de l'exécution des parties séquentielles de l'application.</i>
GPU	est le processeur graphique, « <i>General-Purpose Graphics Processor Unit</i> », pouvant réaliser du travail générique qui peut être utilisé pour implémenter des algorithmes parallèles.
Device	est le GPU connecté à «l'hôte» et qui va exécuter les parties de calcul intensif basé sur un parallélisme de données. <i>Le device, ou périphérique, est responsable de l'exécution de la partie parallèle de l'application.</i>
kernel	est une fonction qui peut être appelée depuis «l'hôte» et qui est exécutée en parallèle sur le «device» CUDA par de nombreuses threads.

- * Le «kernel» est exécuté simultanément par des milliers de threads.

- * Une application ou une fonction de bibliothèque consiste en un ou plusieurs kernels.

Fermi peut exécuter différents kernels à la fois, s'ils appartiennent tous à la même application.

- * Un kernel peut être écrit en C avec des annotations pour exprimer le parallélisme :

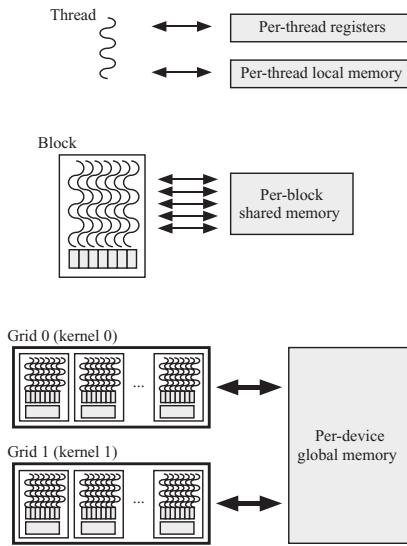
- ◊ localisation des variables ;

- ◊ utilisation d'opération de synchronisation fournie par l'environnement CUDA.



La hiérarchie mémoire

15



La hiérarchie de mémoire et de threads est la suivante :

1. la **thread** au niveau le plus bas de la hiérarchie ;
2. le **bloc** composé de plusieurs threads exécutées de manière concurrente ;
3. la **grille** composée de plusieurs blocs de threads exécutés de manière concurrente ;
4. de la **mémoire locale** dédiée à chaque thread, *per-thread local memory*, visible uniquement depuis la thread (cela concerne également des registres) ;
Les registres sont sur le processeur et disposent de temps d'accès très rapide.
La mémoire locale, indiquée en gris sur le schéma, dispose d'un temps d'accès plus lent que celui des registres.
5. de la **mémoire partagée** associée à chaque bloc visible, *per-block shared memory*, uniquement par toutes les threads du bloc ;
Le bloc dispose de sa propre mémoire partagée et privée pour permettre des communications inter-thread rapides et de taille réglable.
6. de la **mémoire globale**, «*per-device global memory*», utilisable par le «*device*».
Une grille, «grid», utilise la mémoire globale. Cette mémoire globale permet de communiquer avec la mémoire de l'hôte et sert de lien de communication entre l'hôte et le GPGPU.



Grille & Bloc : organisation et localisation d'une thread

Le programmeur doit spécifier le nombre de threads dans un bloc et le nombre de blocs dans une grille.

Le nombre de blocs dans la grille est spécifié par la variable `gridDim`.

Exemple : un tableau à une seule dimension

- ◊ on peut organiser les blocs en un tableau à une seule dimension, et le nombre de blocs sera :

$$\text{gridDim.x} = k$$

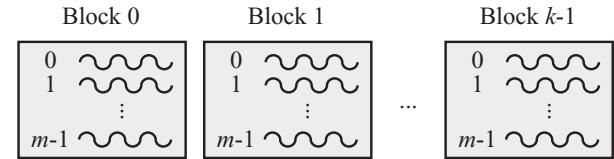
ainsi si $k = 10$, alors on aura 10 blocs dans la grille.

- ◊ on peut organiser les threads en un tableau à une seule dimension de m threads par bloc :

$$\text{blockDim.x} = m$$

- ◊ chaque bloc dispose d'un identifiant unique, «ID», appelé `blockIdx` qui est compris dans l'intervalle :

$$0 \leq \text{blockIdx} \leq \text{gridDim}$$



Pour associer une thread à la $i^{\text{ème}}$ case d'un vecteur, on doit trouver à quel bloc appartient la thread et ensuite la localisation de la thread dans le bloc: $i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$

Généralisation du concept de Grille et de blocs

- ◊ Les variables `gridDim` et `blockIdx` sont définies automatiquement et sont de type `dim3`.
- ◊ Les blocs dans la grille peuvent être organisés suivant une, deux ou trois dimensions.
- ◊ Chaque dimension est accédée par la notation `blockIdx.x`, `blockIdx.y` et `blockIdx.z`.

La commande CUDA suivante définit le nombre de blocs dans les dimensions x , y et z :

```
dim3 dimGrid(4, 8, 1);
```

Cette commande définit 32 blocs organisés en un tableau à deux dimensions avec 4 lignes de 8 colonnes.

Le nombre de threads dans un bloc est défini par la variable `blockDim`.



Chaque thread dispose d'un identifiant unique, «ID», appelé `threadIdx` qui est compris dans l'intervalle :

$$0 \leq \text{threadId} \leq \text{blocDim}$$

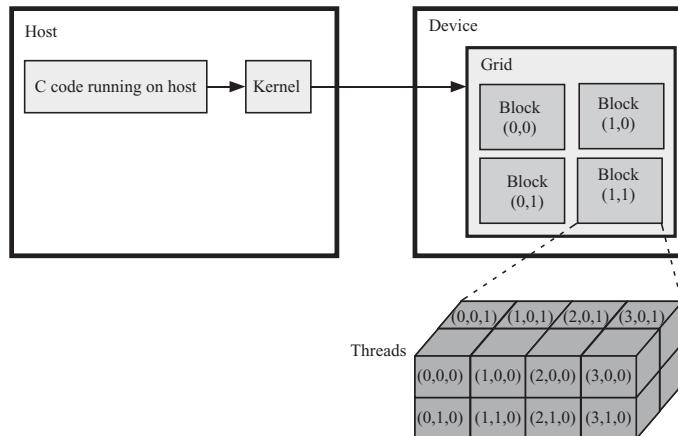
- ◊ Les variables `blockDim` et `threadIdx` sont définies automatiquement et sont de type `dim3`.
- ◊ Les threads dont un bloc peuvent être organisés suivant une, deux ou trois dimensions.
- ◊ Chaque dimension est accédée par la notation `threadIdx.x`, `threadIdx.y` et `threadIdx.z`.

La commande CUDA suivante définit le nombre de threads dans les dimensions *x*, *y* et *z*:

```
dim3 dimBlock(100, 1, 1);
```

Cette commande définit 100 threads organisées en un tableau de 100 cases.

Rapport entre «kernel» et grille



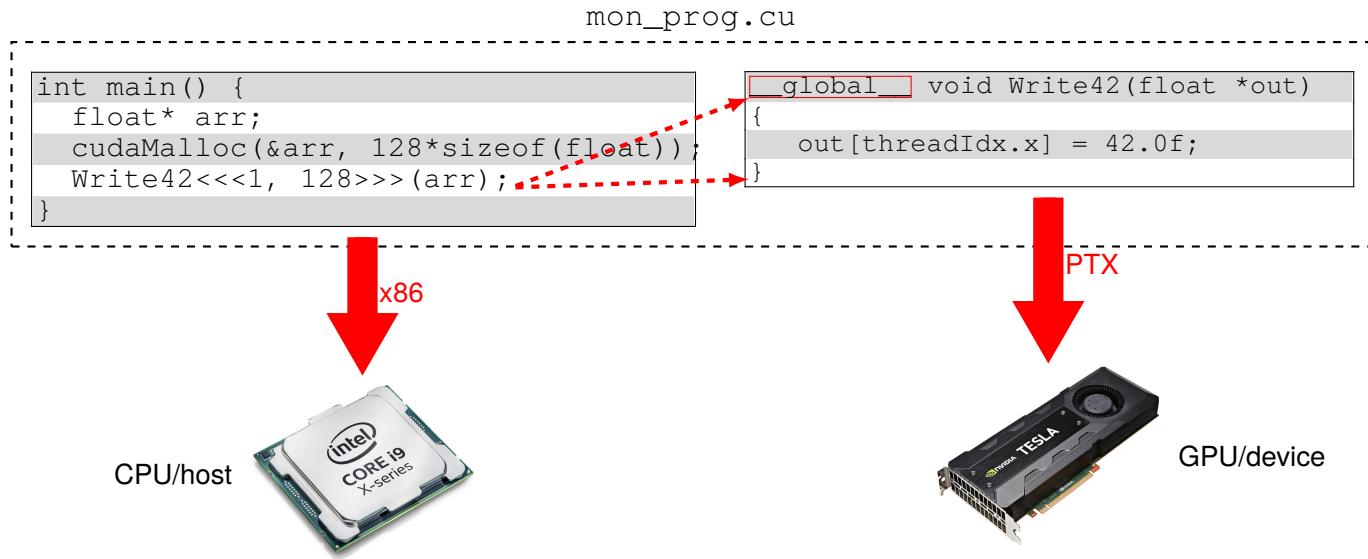
- ◊ Chaque «kernel» est associé avec une grille dans le «device».
- ◊ le choix du nombre de threads et de blocs est conditionné par la nature de l'application et la nature des données à traiter.

Le but est d'offrir au programmeur des moyens d'organiser les threads de manière adaptée à l'organisation des données, afin de simplifier l'accès à ces données : «les données sont organisées en grille ? alors les threads aussi».



Le code source «*mon_prog.cu*» est compilé en deux parties :

- un code pour le CPU en instructions x86/amd64 ;
- un code pour le GPU en instructions PTX, «*Parallel Thread eXecution*» ;



Le compilateur *nvcc* fourni par NVidia :

- ▷ réalise la répartition des codes à partir d'un fichier source unique ;
- ▷ compile chaque partie indépendamment ;
- ▷ construit un exécutable contenant les deux parties et capable de charger le code GPU sur le «*device*».



Pour définir une fonction qui va être exécutée en tant que «kernel», le programmeur modifie le code C du prototype de la fonction en plaçant le mot clé «`__global__`» devant ce prototype :

```
__global__ void kernel_function_name(function_argument_list);
```

La fonction doit renvoyer void.

Le programmeur doit ensuite indiquer au compilateur C NVIDIA, nvcc, de lancer le «kernel» pour être exécuté sur le «device» :

```
int main()
{
/*
    Une partie séquentielle du code
*/
/* Le début de la partie parallèle du code */

kernel_function_name<<< gridDim, blockDim >>> (function_argument_list);

/* La fin de la partie parallèle */

/*
    Une partie séquentielle du code
*/
}
```

Le programmeur modifie le code C en spécifiant la structure des blocs dans la grille et la structure des threads dans un bloc en ajoutant la déclaration `<<<gridDim, blockDim>>>` entre le nom de la fonction et la liste des arguments de la fonction.



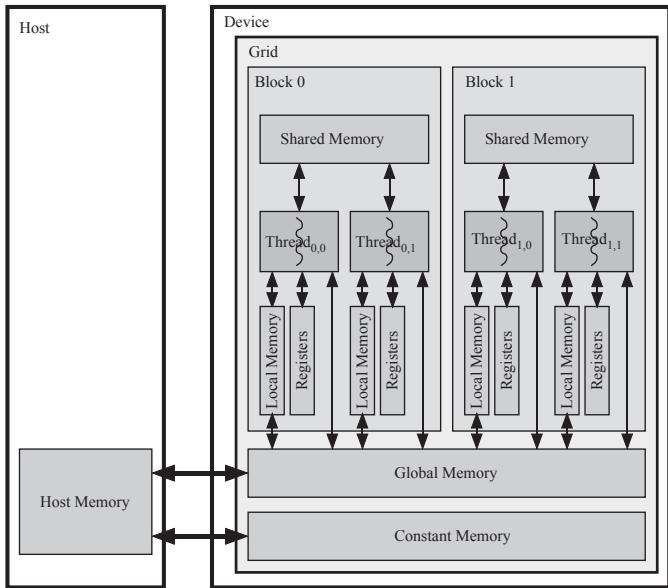
- ▷ L'ordinateur hôte dispose de sa propre hiérarchie de mémoire de même que le «*device*».
- ▷ L'échange de données entre l'hôte et le «*device*» est réalisé en copiant des données entre la DRAM, «dynamic ram», et la mémoire DRAM globale du «*device*».
- ▷ De la même façon qu'en C, le programmeur doit allouer de la mémoire dans la mémoire globale du «*device*» pour les données et libérer cette mémoire une fois l'application terminée.

Les **appels systèmes CUDA** suivants permettent de réaliser ces opérations :

Fonction	Description
cudaDeviceSynchronize ()	bloque jusqu'à ce que le « <i>device</i> » ait terminé les tâches demandées précédemment
cudaThreadSynchronize ()	<i>version précédente de cudaDeviceSynchronize ()</i>
cudaChooseDevice ()	retourne le « <i>device</i> » qui correspond aux propriétés spécifiées
cudaGetDevice ()	retourne le « <i>device</i> » utilisé actuellement
cudaGetDeviceCount ()	retourne le nombre de « <i>device</i> » capable de faire du GPGPU
cudaGetDeviceProperties ()	retourne les informations concernant le « <i>device</i> »
cudaMalloc ()	alloue un objet dans la mémoire globale du « <i>device</i> ». Nécessite deux arguments : l'adresse d'un pointeur qui recevra l'adresse de l'objet, et la taille de l'objet
cudaFree ()	libère l'objet de la mémoire globale du « <i>device</i> »
cudaMemcpy ()	copie des données de l'hôte vers le « <i>device</i> ». Nécessite quatre arguments : le pointeur destination, le pointeur source, le nombre d'octets et le mode de transfert.



L'interface mémoire entre l'hôte et le «device» :



La mémoire globale en bas du schéma est le moyen de communiquer des données entre l'hôte et le «device».

Le contenu de la mémoire globale est visible depuis toutes les threads et est accessible en lecture/écriture, celle indiquée «constant memory», n'est accessible qu'en lecture seulement.

La mémoire partagée par bloc est visible depuis toutes les threads de ce bloc.

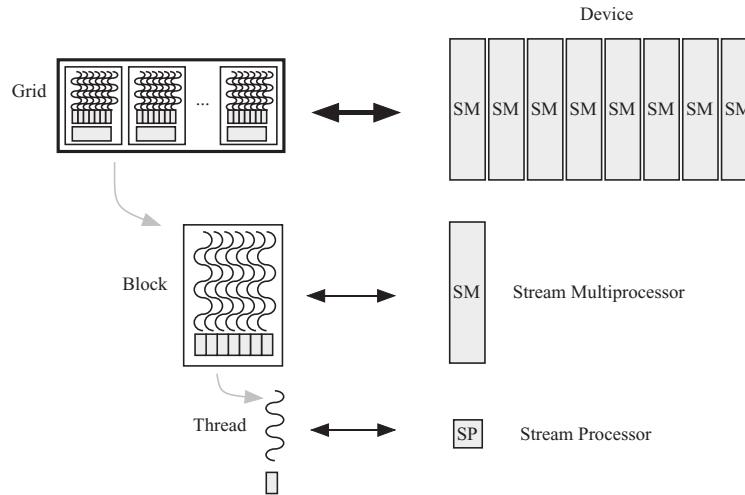
La mémoire locale, comme les registres, n'est visible que de la thread.



Exécution d'une application parallèle sur le «device»

22

L'hôte déclenche une fonction «kernel» :



- ◊ Le «*kernel*» est exécuté sur une grille de blocs de threads.
- ◊ Différents «*kernels*» peuvent être exécutés par le «*device*» à différents moments de la vie du programme.
- ◊ Chaque bloc de threads est exécuté sur un multiprocesseur à flux, «*streaming multiprocessor*», «*SM*».
- ◊ Le *SM* exécute plusieurs blocs de threads à la fois.
- ◊ Des copies du «*kernel*» sont exécutées sur le «*streaming processor*», «*SP*», ou «*thread processors*», ou «*Cuda core*», qui exécute une thread qui évalue la fonction.
- ◊ Chaque thread est allouée à un *SP*.

Actuellement, dans les salles de TP :

- ▷ au plus 1024 threads par dimension du bloc qui communiquent par mémoire partagée ;
- ▷ chaque dimension d'une grille doit être inférieure à 65536 ;
- ▷ la mémoire partagée dans un block $\simeq 16Ko$;
- ▷ la mémoire constante $\simeq 64Ko$;
- ▷ nombre de registres disponibles par block 8192 à 16384.



Comment est-ce que cela se passe dans le GPU ?

23

Utilisation du profiler nvprof

Exemple sur l'exercice du TD n°1 :

```
$ nvprof ./TD1
==21398== NVPROF is profiling process 21398, command: ./TD1
Result : 25723564731392.000000
==21398== Profiling application: ./TD1
==21398== Profiling result:
          Type      Time(%)      Time      Calls      Avg      Min      Max      Name
GPU activities:  84.51%  45.057us           2  22.528us  22.272us  22.785us  [CUDA memcpy HtoD]
                  14.05%  7.4880us           1  7.4880us  7.4880us  7.4880us  dot(double*,,
double*, double*)①
                  1.44%    768ns           1   768ns    768ns    768ns  [CUDA memcpy
DtoH]
API calls:    99.23% 125.79ms           3  41.930ms  6.2840us  125.66ms  cudaMalloc
              0.34% 429.54us          94  4.5690us   524ns  173.12us  cuDeviceGetAttribute
              0.20% 249.19us           3  83.064us  10.325us  121.90us  cudaFree
              0.09% 115.94us           3  38.646us  14.710us  56.079us  cudaMemcpy
              0.08% 103.67us           1  103.67us  103.67us  103.67us  cuDeviceTotalMem
              0.03% 44.132us           1  44.132us  44.132us  44.132us  cuDeviceGetName
              0.02% 24.364us           1  24.364us  24.364us  24.364us  cudaLaunch
              0.00% 2.5540us           3   851ns   532ns  1.3180us  cuDeviceGetCount
              0.00% 1.4740us           2   737ns   590ns   884ns  cuDeviceGet
              0.00%  943ns             1   943ns   943ns   943ns  cudaConfigureCall
              0.00%  930ns             3   310ns   142ns   529ns  cudaSetupArgument

```

Le **profiler** fournit les informations suivantes :

- le **nombre d'appels** des différentes fonctions (sous la rubrique «*Calls*») ;
- le temps d'exécution des **différentes fonctions CUDA** ② ;
- le temps d'exécution du **kernel** (①, le kernel qui ici s'appelle `dot`) ;
- le **rapport** entre le temps d'exécution du **kernel** et des **transferts de mémoire** ③.



Comment est-ce que cela se passe dans le GPU ?

24

Utilisation du profiler nvprof

Ici, on va demander à récupérer des informations concernant les activités du GPU avec l'option --print-gpu-trace:

```
$ nvprof --print-gpu-trace ./TD1
==21538== NVPROF is profiling process 21538, command: ./TD1
Result : 25723564731392.000000
==21538== Profiling application: ./TD1
==21538== Profiling result:
      Start Duration          Grid Size     Block Size   Regs*    SSMem*    DSMem*     Size Throughput
SrcMemType DstMemType           Device       Context     Stream  Name
228.80ms  22.913us           GeForce GTX 106  -        1      -      - [CUDA memcpy HtoD]  -  264.00KB 10.988GB/s
Pageable   Device
228.84ms  22.208us           GeForce GTX 106  -        1      -      - [CUDA memcpy HtoD]  -  264.00KB 11.337GB/s
Pageable   Device
228.87ms  7.4560us          (132 1 1)❶      (256 1)tkzapf2  13❷  2.0000KB  0B [111]  -
-          -           GeForce GTX 106      7      dot(double*, double*, double*) [111]
228.88ms  768ns             GeForce GTX 106  -        1      -      - [CUDA memcpy DtoH]  -  1.0313KB 1.2806GB/s
Device    Pageable
Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

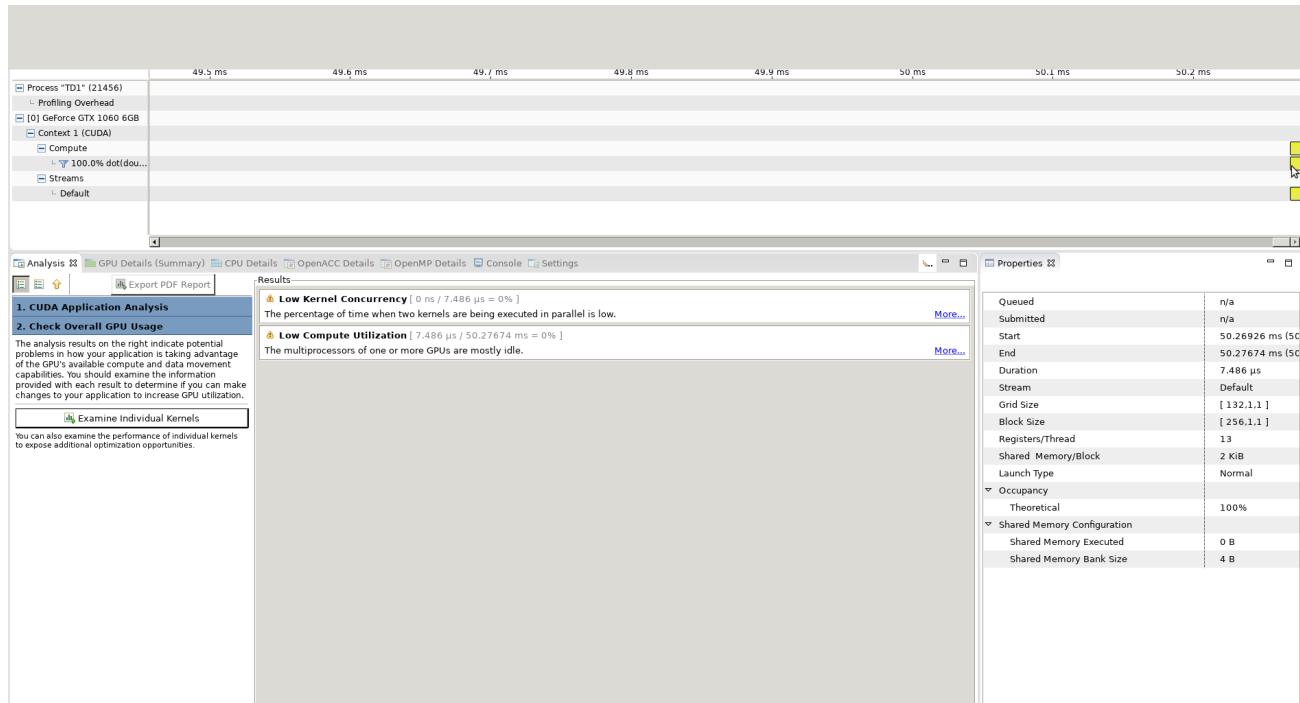
On obtient des informations sur le code PTX, «Parallel Thread eXecution» produit:

- ❶⇒la géométrie de la grille ;
- ❷⇒celle du bloc ;
- ❸⇒le nombre de registres utilisés par thread, c-à-d le nombre de variables locales utilisées par le kernel (si on dépasse, on est obligé d'utiliser de la mémoire locale à la thread moins performante).



Comment est-ce que cela se passe dans le GPU ?

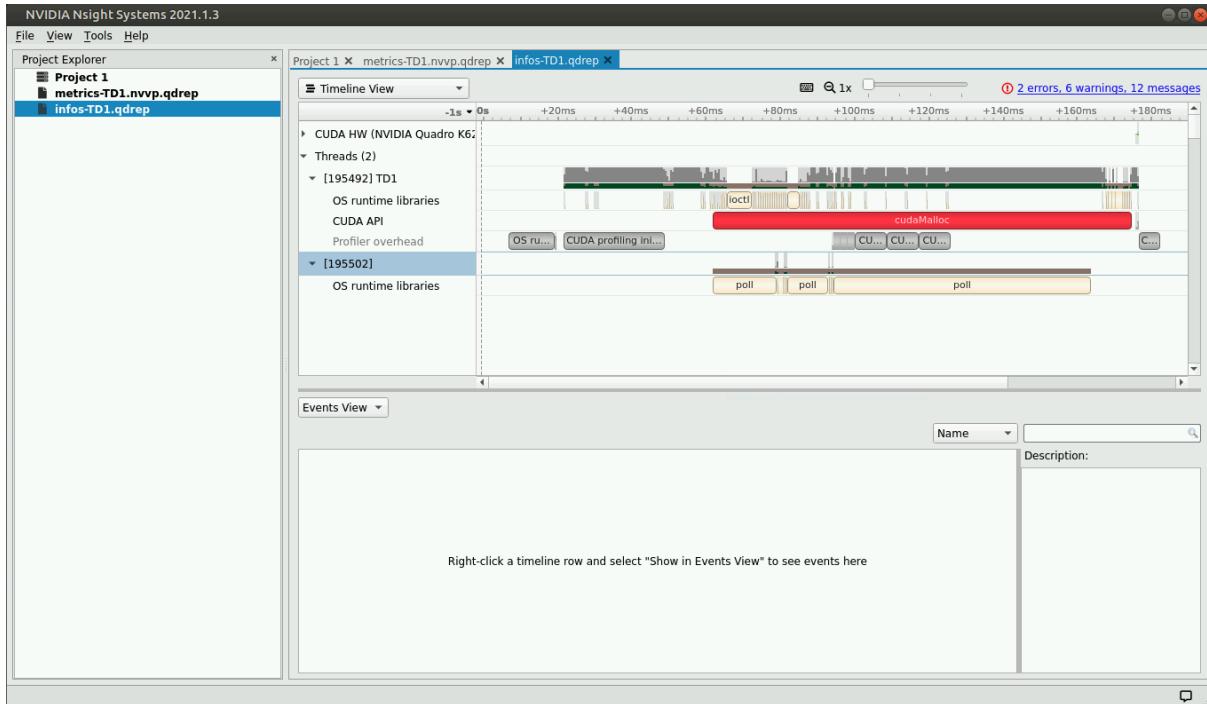
25



Une copie d'écran de l'outil «nvvp».

Comment est-ce que cela se passe dans le GPU ?

26



Une copie d'écran de l'outil «nsight».

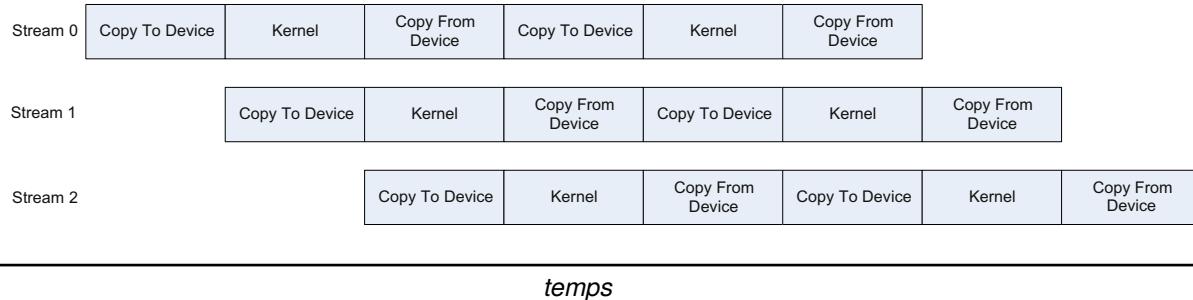
La commande utilisée :

```
└── xterm └──
$ nsy profile -w true -t cuda,nvtx,osrt,cudnn,cublas -s cpu --cudabacktrace=true -x true -o
infos-TD1 ./TD1
```



La notion de «stream»

- sur une carte pro : plusieurs streams possibles ;
- sur une carte grand public : un seul stream.



Ici, la carte GPGPU est capable de supporter plusieurs streams, mais offre un accès séquentielle pour les transferts des données de l'hôte vers la carte, «Copy To device».

Attention

La possibilité de faire des transferts **asynchrones**, c-à-d de «recouvrir» des communications par du calcul est obtenu à l'aide de la fonction `cudaMemcpyAsync()` :

- ▷ la copie de mémoire **vers le GPGPU** depuis le CPU peut être réalisé en **même temps** que du travail sur le CPU (le GPU récupère ses données simultanément) ;
- ▷ la copie **vers et depuis** le GPU peut être faire pendant que le GPU réalise du travail.

Cette capacité est disponible suivant la capacité CUDA de la carte nvidia utilisée.



Soit le programme suivant et sa parallélisation :

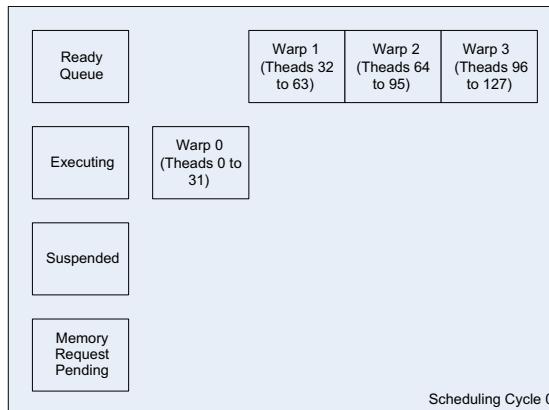
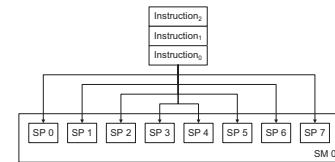
```
void some_func(void)
{
    int i;
    for (i=0;i<128;i++)
        { a[i] = b[i] * c[i]; }
}
```

On parallélise la boucle en créant une thread par occurrence de la boucle :

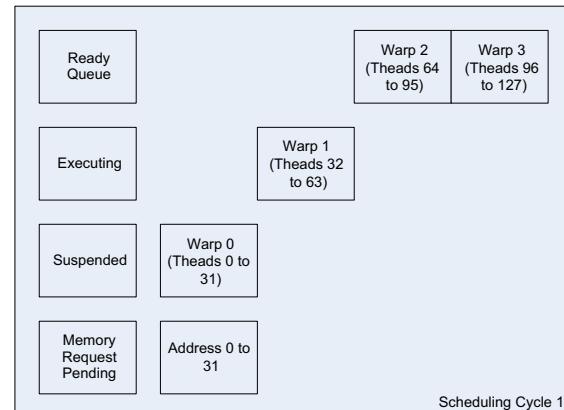
```
__global__ void some_kernel_func(int *a, int *b, int *c)
{   unsigned int thread_idx = threadIdx.x;
    a[thread_idx] = b[thread_idx] * c[thread_idx]; }
```

Un «warp»

- ▷ correspond à 32 threads ;
- ▷ exécute du code SPMD, ou SPMT, «*Simple Program Multiple Thread*»,
- ▷ est ordonné, *scheduled*, dans le SP, suivant son état :



Le «Warp 0» progresse de «Ready Queue» à «Executing».



Le «Warp 0» réalise des demandes d'accès mémoire et se suspend : c'est le «Warp 1» qui s'exécute.

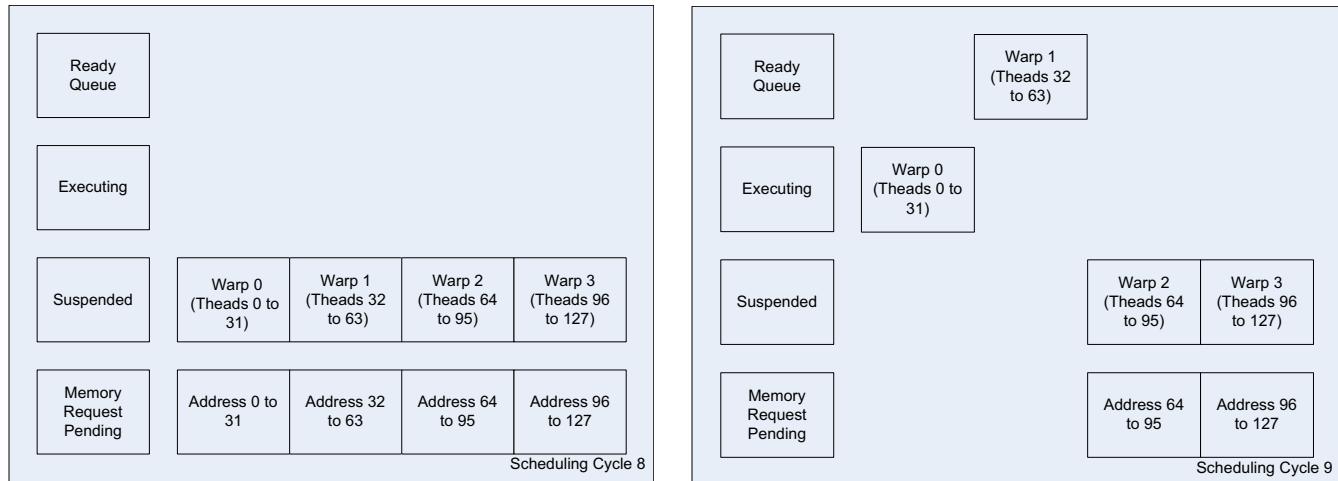


La notion de «warp»

29

Le «scheduler» fait progresser le warp de l'état prêt, «Ready Queue», à l'état exécuté, «Executing».

Dans le cas où le warp réclame du contenu dans la mémoire : il passe en «Suspended» et des accès mémoires attendent d'être résolus : «Memory Request Pending».



Toutes les threads sont bloquées en attente du retour des données depuis la mémoire...

Les données 0 à 63 sont obtenues : les threads 0 à 31 sont exécutées et les threads 32 à 63 sont prêtes.



La notion de divergence

```
__global__ some_func(void)
{
    if (some_condition)
    {
        action_a(); // +
    }
    else {
        action_b(); // -
    }
}
```

Imaginons que les threads paires réalisent le travail positif et les threads impaires le travail négatif par rapport à la condition :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+

Le Warp exécute du code SPMT : les threads «+» s'exécutent pendant que les autres sont bloquées.

Une solution : l'association par demi warp, soient 16 threads :

```
if ((thread_idx % 32) < 16)
{
    action_a();
}
else {
    action_b();
}
```

On bénéficie

- * d'une exécution parallèle de la partie «+» et «-» sur chacun des demi-warps en SPMT ;
- * éventuellement d'accès mémoire sur $4 * 16 = 64$ octets pour les données sur 32bits utilisées par le demi-warp ;



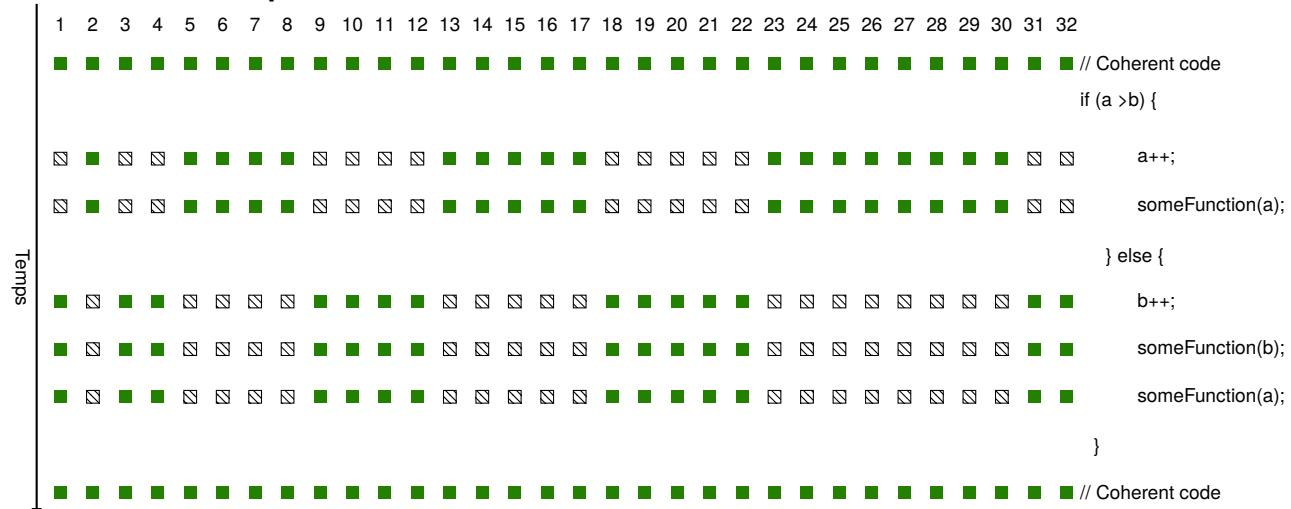
3 La notion de divergence

31

Soit le code suivant:

```
// Coherent code
if (a >b) {
    a++;
    someFunction(a);
} else {
    b++;
    someFunction(b);
    someFunction(a);
}
// Coherent Code
```

Les effets sur le Warp



⇒ Le travail des threads est le même : elles font toutes le travail des deux branches mais on annule le résultat du travail inutile.



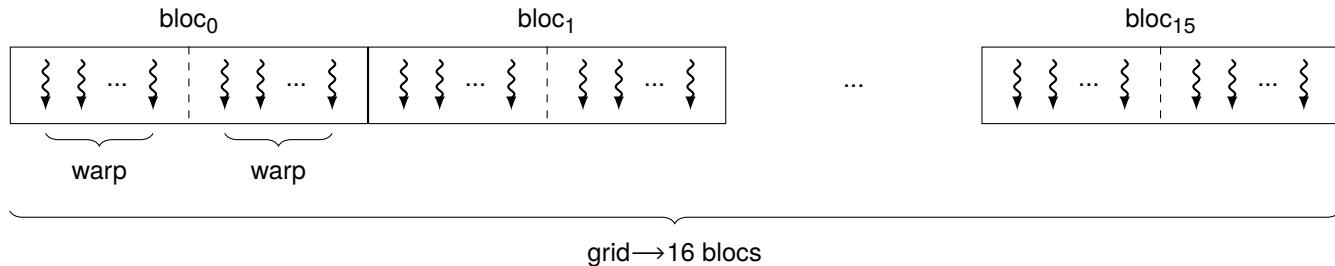
Exemple de code

```
#define WARP_SIZE 32
#define BLOCK_SIZE 2*WARP_SIZE
#define GRID_SIZE 16

...
dim3 dimBlock(BLOCK_SIZE);
dim3 dimGrid(GRID_SIZE);
...
divergence<<<dimGrid, dimBlock>>>(ref_a, ref_b);
```

Dans le kernel, la numérotation de la thread est similaire à l'index du tableau de données :

```
int a = blockIdx.x*blockDim.x + threadIdx.x;
```

Répartition du code entre les blocs et les threads

- ▷ chaque bloc dispose de 64 threads, soient 2 Warps ;
- ▷ il y a 16 blocs dans la grille.



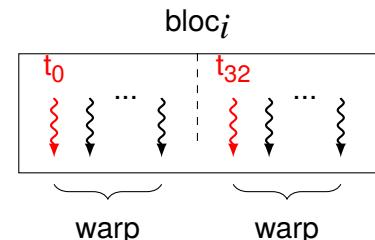
Comparaison de performance divergence/pas de divergence

33

Exemples de Kernels : «divergence» et «noDivergence»

```
__global__ void divergence(float *A, float *B){  
    int a = blockIdx.x*blockDim.x + threadIdx.x;  
    if((threadIdx.x % WARP_SIZE) == 0)  
    {  
        for(int i=0;i<ITERATIONS;i++) {  
            B[a]=A[a]+1;  
        }  
    } else  
        for(int i=0;i<ITERATIONS;i++) {  
            B[a]=A[a]-1;  
        }  
}
```

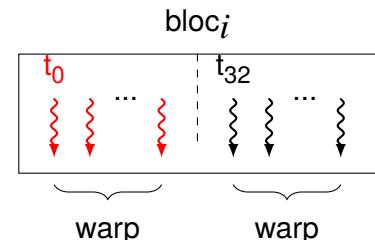
th₀, th₃₂
autres threads



Une seule thread effectue un travail différent (la première thread du warp, en rouge sur le schéma).

```
__global__ void noDivergence(float *A, float *B){  
    int a = blockIdx.x*blockDim.x + threadIdx.x;  
    if(threadIdx.x >= WARP_SIZE)  
    {  
        for(int i=0;i<ITERATIONS;i++) {  
            B[a]=A[a]+1;  
        }  
    } else  
        for(int i=0;i<ITERATIONS;i++) {  
            B[a]=A[a]-1;  
        }  
}
```

th₃₂, th₃₃ à th₆₃
th₀, th₁ à th₃₁



Un warp complet réalise chaque branche de la condition (en rouge sur le schéma).



Au niveau de l'exécution

Sans divergence : un warp complet exécute une des branches de la condition.

```
└── xterm └──
pef@fpga:~/CUDA/MESURES_PERFS$ ./divergence
kernel invocation
Sans divergence
kernel execution time (msecs): 154.822556 ms
32 -> 1.000000.0 33 -> 1.000000.0 34 -> 1.000000.0 35 -> 1.000000.0 36 ->
1.000000.0 37 -> 1.000000.0 38 -> 1.000000.0 39 -> 1.000000.0 40 -> 1.000000.0
41 -> 1.000000.0 42 -> 1.000000.0 43 -> 1.000000.0 44 -> 1.000000.0 45 ->
1.000000.0 46 -> 1.000000.0 47 -> 1.000000.0 48 -> 1.000000.0 49 -> 1.000000.0
50 -> 1.000000.0 51 -> 1.000000.0 52 -> 1.000000.0 53 -> 1.000000.0 54 ->
1.000000.0 55 -> 1.000000.0 56 -> 1.000000.0 57 -> 1.000000.0 58 -> 1.000000.0
59 -> 1.000000.0 60 -> 1.000000.0 61 -> 1.000000.0 62 -> 1.000000.0 63 ->
1.000000.0 96 -> 1.000000.0 97 -> 1.000000.0 98 -> 1.000000.0 99 -> 1.000000.0
100 -> 1.000000.0 101 -> 1.000000.0 102 -> 1.000000.0 103 -> 1.000000.0 104 ->
1.000000.0 105 -> 1.000000.0 106 ->
...

```

Sans divergence : un thread par warp introduit une divergence :

```
└── xterm └──
pef@fpga:~/CUDA/MESURES_PERFS$ ./divergence yes
kernel invocation
Avec divergence
kernel execution time (msecs): 290.697205 ms
0 -> 1.000000.0 32 -> 1.000000.0 64 -> 1.000000.0 96 -> 1.000000.0 128 ->
1.000000.0 160 -> 1.000000.0 192 -> 1.000000.0 224 -> 1.000000.0 256 ->
1.000000.0 288 -> 1.000000.0 320 -> 1.000000.0 352 -> 1.000000.0 384 ->
1.000000.0 416 -> 1.000000.0 448 -> 1.000000.0 480 -> 1.000000.0 512 ->
1.000000.0 544 -> 1.000000.0 576 -> 1.000000.0 608 -> 1.000000.0 640 ->
1.000000.0 672 -> 1.000000.0 704 -> 1.000000.0 736 -> 1.000000.0 768 ->
1.000000.0 800 -> 1.000000.0 832 -> 1.000000.0 864 -> 1.000000.0 896 ->
1.000000.0 928 -> 1.000000.0 960 -> 1.000000.0 992 -> 1.000000.0

```

⇒ Les performances vont du simple au double !



Lorsqu'un programme parallèle est exécuté sur le «device», la synchronisation et les communications parmi les threads doivent être réalisées à différents niveaux :

1. «Kernels» et «grids» ;
2. Blocs ;
3. Threads.

Grille & Kernels

Différents «kernels» peuvent être exécutés sur le «device».

```
void main() {  
...  
kernel_1<<<nblocks_1, blocksize_1>>>(fonction_argument_liste_1)  
kernel_2<<<nblocks_2, blocksize_2>>>(fonction_argument_liste_2);  
...}
```

- * kernel_1 va être exécuté en premier sur le «device» :
 - ◊ il va définir une grille qui contiendra dimGrid blocs, chacun de ces blocs contiendra dimBlock threads.
 - ◊ toutes les threads vont exécuter le même code spécifié par le «kernel».
- * lorsque kernel_1 aura terminé, alors kernel_2 sera transmis vers le «device» pour son exécution.

Attention

Le communication entre les différentes grilles est **indirecte** : elle consiste à laisse en place les données dans l'hôte ou la mémoire globale du «device» pour être utilisées par le prochain «kernel».



Les blocs

- * Tous les blocs d'une grille s'exécutent indépendamment les uns des autres : il n'y a **pas de mécanisme de synchronisation** entre les blocs.
- * lorsqu'une grille est lancée, les blocs sont assignés à un SM, dans un **ordre arbitraire** qui n'est pas **prédictible**.

Les communications entre les threads à l'intérieur d'un bloc sont réalisées au travers de la **mémoire partagée du bloc** :

- ◊ une variable est déclarée comme étant partagée par les threads du même bloc en préfixant sa déclaration à l'aide du mot-clé «__shared__».
Cette variable est alors stockée dans la mémoire partagée du bloc.
- ◊ lors de l'exécution d'un «kernel», une version privée de cette variable est créée dans la mémoire locale de la thread.

Des temps d'accès mémoire différents

- ◊ La **mémoire partagée** associée au bloc est sur la même puce que les **coeurs**, «cores», exécutant les threads ;
La communication est relativement rapide, car la SRAM, «*Static RAM*», est plus rapide que la mémoire située en dehors de la puce, «off-chip» de type DRAM ;
- ◊ chaque thread dispose d'un **accès direct à ses registres** inclus dans la puce et d'un accès direct à sa mémoire locale qui est en «off-chip». *Les registres sont beaucoup plus rapides que la mémoire locale* ;
- ◊ chaque thread peut également avoir **accès à la mémoire globale** du «device» ;
- ◊ l'accès d'une thread à la mémoire locale et globale souffre des problèmes inhérents aux communications entre puces, «*interchip*» : *retard, consommation de puissance, débit*.



Les threads et le SIMD : la notion de «warp»

Un grand nombre de threads sont exécutées sur le «device».

Un bloc qui est assigné à un SM est divisé en groupe de 32 threads *warps*. Chaque warp représente le «SIMD» du GPU.

Chaque SM peut gérer plusieurs «warps» simultanément, et lorsque certains «warps» sont bloqués à cause d'accès à la mémoire, le SM peut ordonner l'exécution d'un autre «warp» (suivant un scheduler).

- ◊ Les threads d'un **même bloc** peuvent se synchroniser à l'aide de la fonction `__syncthreads()`, réalisant une barrière de synchronisation entre les différents warps exécutés.
- ◊ une thread peut utiliser une opération **atomique** pour obtenir l'accès exclusif à une variable pour un réaliser une opération donnée: `atomicAdd(result, input[threadIdx.x])` coûteux en synchronisation.
- ◊ Chaque thread utilise ses registres et sa mémoire locale, qui utilise tous les deux de la SRAM, ce qui induit une petite quantité de mémoire disponible, mais des communications rapides et peut coûteuse en énergie.
- ◊ Chaque thread peut également utiliser la mémoire globale qui est plus lente car elle utilise de la DRAM.

La répartition des variables entre les différentes zones mémoire

Pour définir la localisation, on utilise des préfixe pour la déclaration ou des règles automatiques.

Déclaration	Lieu de stockage de la variable	Pénalité	Portée	Lifetime
<code>int var;</code>	un registre de la thread		thread	thread
<code>int tableau[10];</code>	la mémoire locale de la thread		thread	thread
<code>__shared__ int var;</code>	la mémoire partagée du bloc		bloc	bloc
<code>__device__ int var;</code>	la mémoire globale du «device»	<i>plus lent</i>	grille	application
<code>__constant__ int const;</code>	la mémoire constante du bloc.		grille	application

Les variables `__constant__` et `__device__` sont à déclarer en dehors de toute fonction.

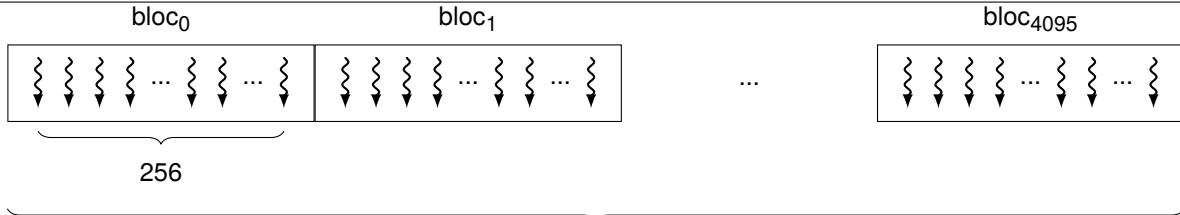


Des accès décalés ou répartis

permet de choisir float ou double

```
template <typename T> __global__ void offset(T* a, int s)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x + s;
    a[i] = a[i] + 1;
}
template <typename T> __global__ void stride(T* a, int s)
{
    int i = (blockDim.x * blockIdx.x + threadIdx.x) * s;
    a[i] = a[i] + 1;
}
```

```
int blockSize = 256; // taille du bloc
...
int n = nMB*1024*1024/sizeof(T); // si nMB = 4, n=1048576
...
checkCuda( cudaMalloc(&d_a, n * 33 * sizeof(T)) ); // 33*1024*1024 var. type T
...
offset<<<n/blockSize, blockSize>>>(d_a, i); // la grille est de 4096 blocs
...
if (bFp64) runTest<double>(deviceId, nMB); // on utilise des doubles
else        runTest<float>(deviceId, nMB); // ou des floats
```



Décalage des accès ou «offset»

On décale 32 fois :

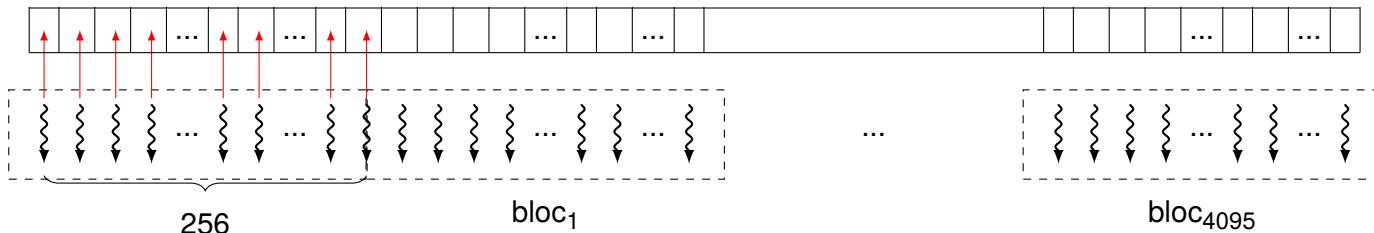
```
for (int i = 0; i <= 32; i++) {
    offset<<<n/blockSize, blockSize>>>(d_a, i);
    ...
    printf("%d, %f\n", i, 2*nMB/ms); // on mesure le débit
}
```

Le travail du Kernel :

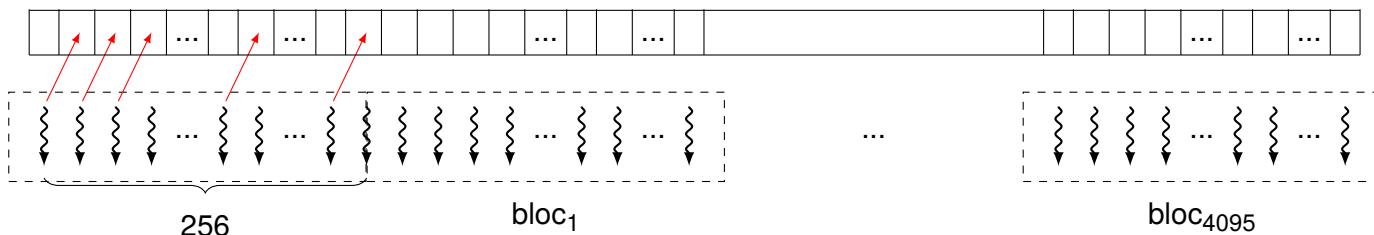
```
__global__ void offset(T* a, int s)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x + s;
    a[i] = a[i] + 1;
}
```

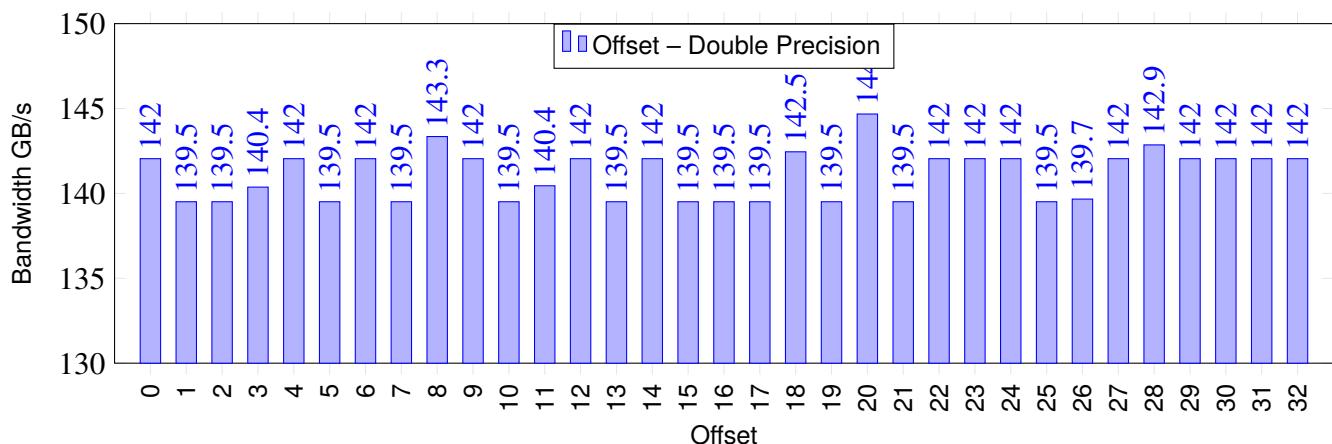
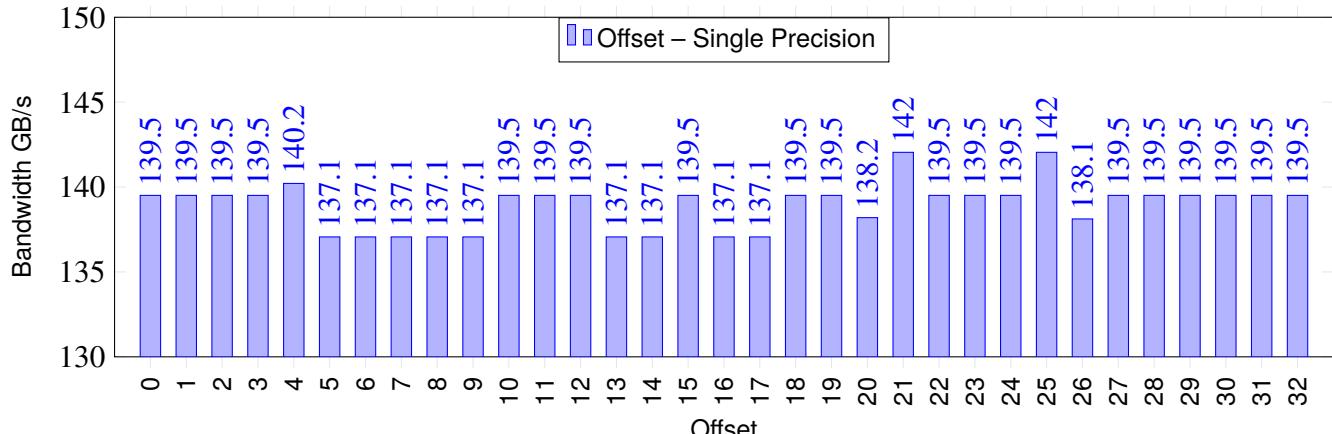
addition

Offset de zéro



Offset de un



Aggrégation, «*coalescence*», des accès mémoire : décalage

Saut des accès ou «stride»

On accède à 2 fois, puis 3 fois, etc:

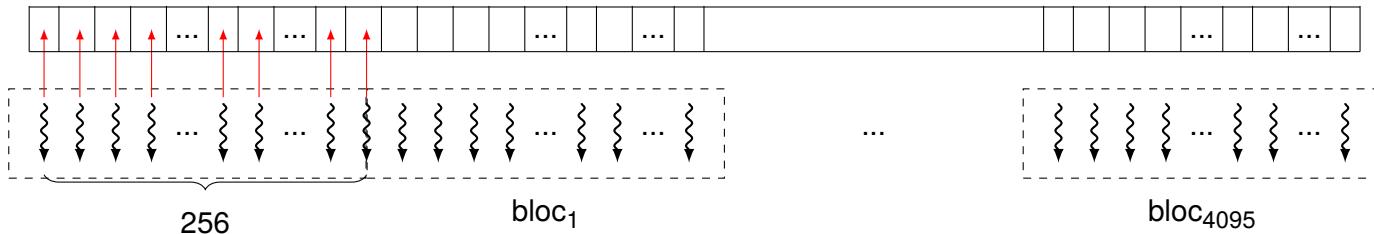
```
for (int i = 0; i <= 32; i++) {
    stride<<<n/blockSize, blockSize>>>(d_a, i);
    ...
    printf("%d, %f\n", i, 2*nMB/ms); // on mesure le débit
}
```

Le travail du Kernel :

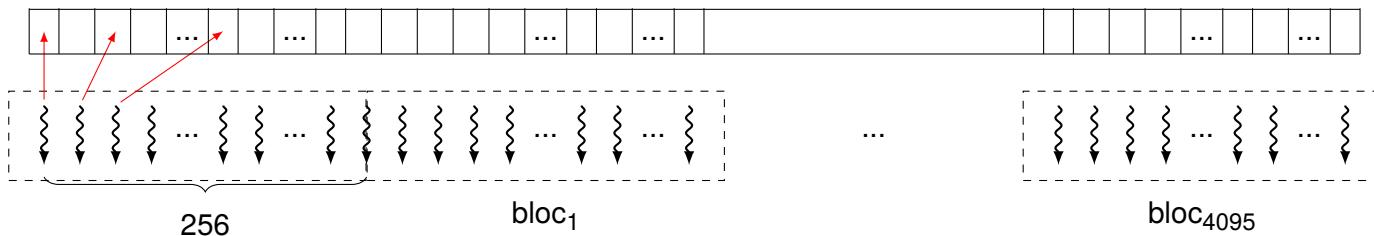
```
__global__ void offset(T* a, int s)
{
    int i = blockDim.x * blockIdx.x
            + threadIdx.x * s;
    a[i] = a[i] + 1;
}
```

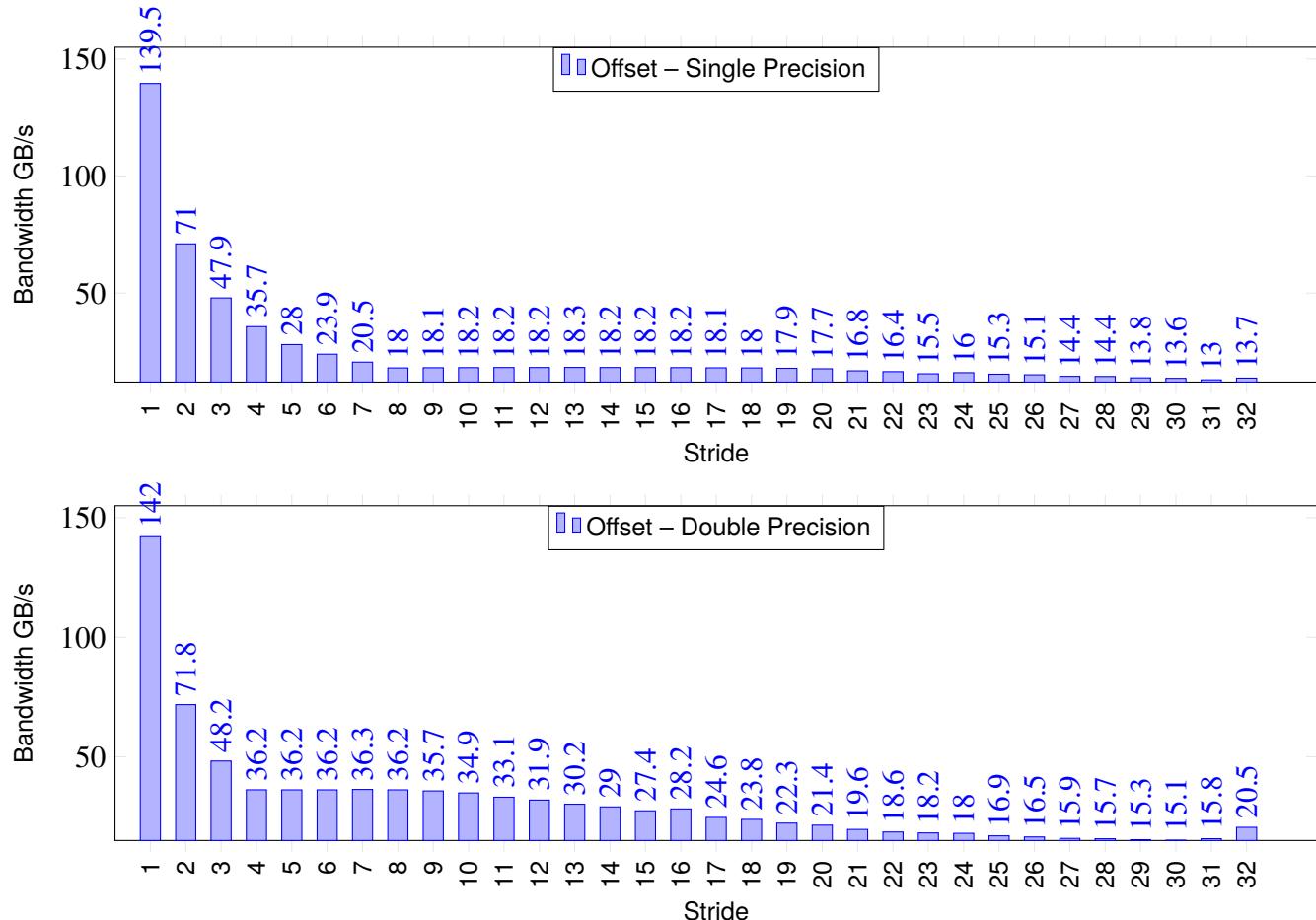
multiplication

Saut de zéro



Saut de un





Exemple de code utilisant la «*shared memory*»

```
// Calcul de différence de données adjacentes
// calculer result[i] = input[i] - input[i-1]
__device__ int result[N];

__global__ void adj_diff_naive(int *result, int *input)
{
    // calculer l'identifiant de la thread en fonction
    // de sa position dans la grille
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // Utiliser des variables locales à la thread
        int x_i = input[i];
        int x_i_minus_one = input[i-1];
        // Deux accès sont nécessaires pour i et i-1
        result[i] = x_i - x_i_minus_one;
    }
}

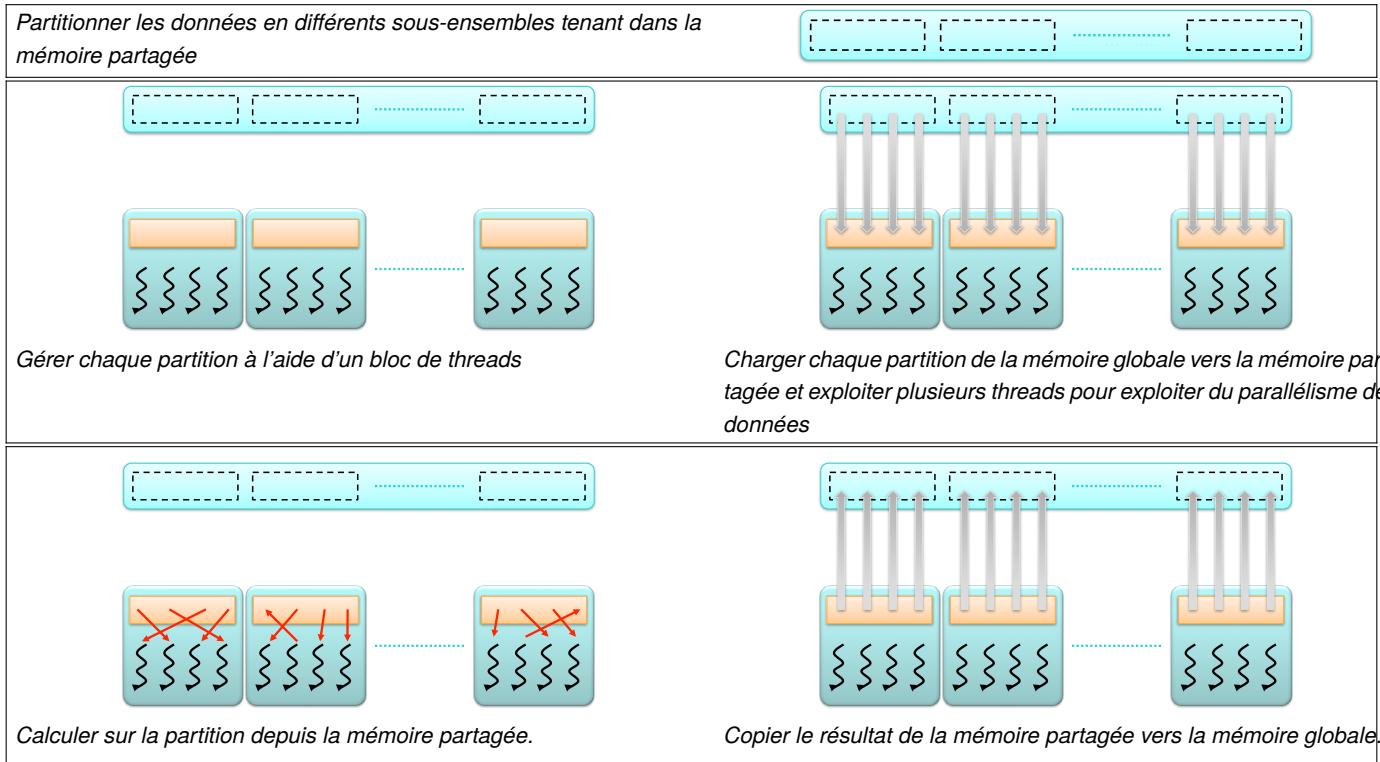
// version optimisée
__device__ int result[N];
__global__ void adj_diff(int *result, int *input)
{
    // raccourci pour threadIdx.x
    int tx = threadIdx.x;
    // allouer un __shared__ tableau,
    // un élément par thread
    __shared__ int s_data[BLOCK_SIZE];
    // chaque thread lit un élément dans s_data
    unsigned int i = blockDim.x * blockIdx.x
                    + tx;
    s_data[tx] = input[i];

    // éviter une race condition: s'assurer
    // des chargements avant de continuer
    __syncthreads();
    if(tx > 0)
        result[i] = s_data[tx] - s_data[tx-1];
    else if(i > 0)
    { // traiter les accès aux bords du bloc
        result[i] = s_data[tx] - input[i-1];
    }
}
```

Cette optimisation se traduit par une nette amélioration des performances : 36,8GB/s vs 57.5GB/s.



- * la **mémoire globale** réside dans la mémoire globale du «device» en DRAM (plus lente) ;
- * il faut **partitionner les données** pour tirer parti de la **mémoire partagée** plus rapide :
 - ◊ utiliser la technique vu sur le transparent précédent ;
 - ◊ utiliser une méthode «divide and conquer»



Dessiner un ensemble de Julia

On parcourt une surface 2D :

- * pour chaque point de coordonnées (x, y) , on traduit ses coordonnées dans l'espace complexe :
 - ◊ Soit $DIM \times DIM$ la dimension de la surface en pixels ;
 - ◊ on centre cet espace complexe au centre de l'image, en décalant de $DIM/2$;
 - ◊ ce qui donne pour un point (x, y) , les coordonnées

$$((DIM/2 - x)/(DIM/2), (DIM/2 - y)/(DIM/2))$$

- * on utilise également un «zoom» avec la variable *scale* pour zoomer ou dézoomer sur l'ensemble de Julia ;
- * la constante $C = -0.8 + 1.5i$ donne une image intéressante.
- * on considère le complexe $Z_0 = ((DIM/2 - x)/(DIM/2)) + ((DIM/2 - y)/(DIM/2))i$
- * on calcule alors pour ce complexe Z_0 , la limite de la suite $Z_{n+1} = Z_n^2 + C$:
 - ◊ si la limite de la suite diverge ou tend vers l'infini, alors le point n'appartient pas à l'ensemble et il sera coloré en noir ;
 - ◊ si la limite de la suite ne diverge pas, alors le point fait partie de l'ensemble et sera coloré en rouge.
- * pour la limite, on utilise une simplification : si après 200 itérations de calcul de la suite, la valeur au carré de la suite est toujours inférieure à 1000 alors on considère qu'elle ne diverge pas.



Un exemple complet

46

Version CPU

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM ); // DIM = 1000
    unsigned char *ptr = bitmap.get_ptr();
    kernel( ptr );
    bitmap.display_and_exit();
}
```

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);
    int i = 0;
    for (i=0; i<200; i++) { // Calcul de la limite
        a = a * a + c;
        if (a.magnitude2() > 1000) // Infini ?
            return 0; // Pas dans l'ensemble
    }
    return 1; // Dans l'ensemble
}
```

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM; // 2D -> 1D

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue; // Red
            ptr[offset*4 + 1] = 0; // Green
            ptr[offset*4 + 2] = 0; // Blue
            ptr[offset*4 + 3] = 255; } // Alpha
    }
}
```

```
struct cuComplex { // Opérations sur les complexes
    float r, i;

    cuComplex( float a, float b ) : r(a), i(b) {}

    float magnitude2(void) { return r * r + i * i; }

    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r-i*a.i, i*a.r+r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```



Un exemple complet

47

Version GPU

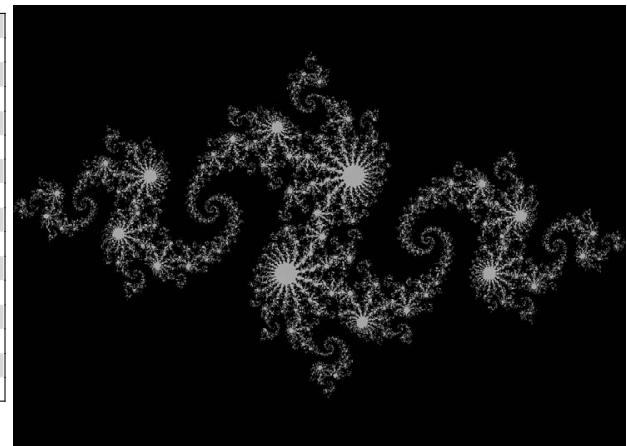
```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) ); // On alloue la zone de l'image
    dim3 grid(DIM,DIM); // On utilise dim3 même si la dernière coordonnée vaudra seulement 1
    kernel<<<grid,1>>>( dev_bitmap ); // Un bloc par pixel et une thread par bloc
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap, bitmap.image_size(),
                           cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();
    cudaFree( dev_bitmap );
}

__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = blockIdx.x; // Calculé automatiquement
    int y = blockIdx.y; // Calculé automatiquement
    int offset = x + y * gridDim.x;
    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}

__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);
    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);
    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }
    return 1;
}
```



```
struct cuComplex {  
    float r, i;  
    cuComplex( float a, float b ) : r(a), i(b) {}  
    __device__ float magnitude2( void )  
    {  
        return r * r + i * i;  
    }  
    __device__ cuComplex operator*(const cuComplex& a)  
    {  
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);  
    }  
    __device__ cuComplex operator+(const cuComplex& a)  
    {  
        return cuComplex(r+a.r, i+a.i);  
    }  
};
```



Les fonctions préfixées par `__device__`:

- ◊ tournent sur le GPU;
- ◊ ne peuvent être appelées que depuis une fonction préfixée par `__device__` ou `__global__`.

Le code complet est accessible sur <http://developer.nvidia.com/cuda-cc-sdk-code-samples>

Parcours imbriqué d'un tableau en version CPU

```
void add_matrix ( float* a, float* b, float* c, int N )
{
    int index;
    for ( int i = 0; i < N; ++i )
        for ( int j = 0; j < N; ++j ) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}
int main() {
    add_matrix( a, b, c, N );
}
```

Ici, la matrice est définie suivant un tableau à une seule dimension.

Parcours imbriqué d'un tableau en version GPU

```
__global__ add_matrix ( float* a, float* b, float* c,
int N )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}
int main() {
    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

On «déplie» les deux boucles imbriquées en une grille, où chaque élément de la grille définit une combinaison de valeurs pour i et j .



```
const int N = 1024;
const int blocksize = 16;
__global__ void add_matrix( float* a, float *b, float *c, int N )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( (i < N) && (j < N) ) c[index] = a[index] + b[index];
}

int main() {
    float *a = new float[N*N]; float *b = new float[N*N]; float *c = new float[N*N];
    for ( int i = 0; i < N*N; ++i ) { a[i] = 1.0f; b[i] = 3.5f; }
    float *ad, *bd, *cd;
    const int size = N*N*sizeof(float);
    cudaMalloc( (void**)&ad, size );
    cudaMalloc( (void**)&bd, size );
    cudaMalloc( (void**)&cd, size );

    cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

    cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

    cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
    delete[] a; delete[] b; delete[] c;

    return EXIT_SUCCESS;
}
```



Les contraintes d'architecture

- ▷ un GPU possède N multiprocesseur, MP , en général 2 ou 4 ;
- ▷ chaque MP possède M processeur scalaire, SP ;
- ▷ chaque MP traite des blocks par lot :
 - ◊ un bloc est traité par un MP uniquement ;
- ▷ chaque bloc est découpé en groupe de threads SIMD appelé warp :
 - ◊ un warp est exécuté physiquement en parallèle ;
- ▷ le scheduler « switche » entre les warps ;
- ▷ un warp contient des threads d'identifiant consécutifs, «thread ID» ;
- ▷ la taille d'un warp est actuellement de 32 ;

Les optimisations possibles

- ▷ $\frac{\text{nombre de blocks}}{\text{nombre de MP}} > 1 \Rightarrow$ tous les MPs ont toujours quelque chose à faire ;
- ▷ $\frac{\text{nombre de blocks}}{\text{nombre de MP}} > 2 \Rightarrow$ plusieurs blocs peuvent être exécutés en parallèle sur un MP ;
- ▷ les ressources par block \leq au total des ressources disponibles :
 - ◊ mémoire partagée et registres ;
 - ◊ plusieurs blocs peuvent être exécutés en parallèle sur un MP ;
 - ◊ éviter les **branchements divergents** dans les conditions à l'intérieur d'un bloc :
 - * les différents chemins d'exécution doivent être **sérialisés** !



Une grille 1D avec des blocs en 2D

```
UniqueBlockIndex = blockIdx.x;  
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;
```

Une grille 1D avec des blocs en 3D

```
UniqueBlockIndex = blockIdx.x;  
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y * blockDim.z  
+ threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

Une grille 2D avec des blocs 1D

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;  
UniqueThreadIndex = UniqueBlockIndex * blockDim.x + threadIdx.x;
```

Un grille 2D avec des blocs 2D

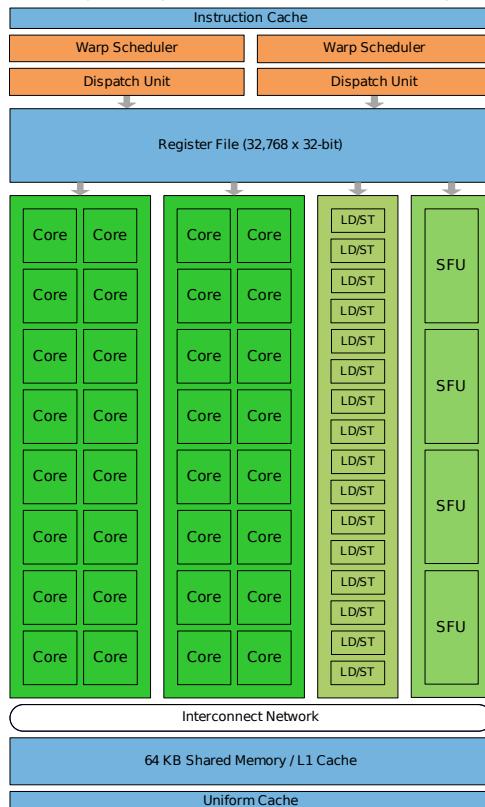
```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;  
UniqueThreadIndex = UniqueBlockIndex * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

Une grille 2D avec des blocs 3D

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;  
UniqueThreadIndex = UniqueBlockIndex * blockDim.z * blockDim.y * blockDim.x  
+ threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```



- «*GigaThread global scheduler*» :
- ◊ distribue les blocs de threads aux SMs ;
- ◊ gère le changement de contexte entre les threads pendant l'exécution ;
- DRAM : jusqu'à 6GB de mémoire grâce à un adressage sur 64bits, débit de 192GB/s ;
- Fréquence processeur : 1,5GHz, Peak performance : 1,5TFlops, Fréquence mémoire : 2GHz ;



- un «core» CUDA :
- ◊ une ALU, «*Integer Arithmetic Logic Unit*» : supporte des opérations sur 32bits, peut également travailler sur 64bits ;
- ◊ FPU, «*Floating Point Unit*» : réalise des «*Fused Multiply-Add*», $A * B + C$, jusqu'à 16 opérations en double précision par SM et par cycle ;
- LoaD/STore : calcule les adresses sources et destinations pour 16 threads par cycle d'horloge depuis/vers la DRAM ou le cache ;
- SM, «*Streaming Multi-processors*» contient :
 - ◊ 32 cœurs «*single precision*» ;
 - ◊ 4 unités SFUs, «*Special Function Units*» :
 - * fonctions transcendentales sinus/cosinus, inverse et racine carrée ;
 - * une instruction par thread et par cycle d'horloge : un «*warp*» exécute l'opération en 8 cycles ;
 - ◊ un bloc de 64KB de mémoire rapide : mémoire cache L1
 - ◊ 32k of 32 bits registers :
 - * chaque thread possède ses propres registres entre 21 et 63 (l'augmentation du nombre de threads diminue le nombre de registres par threads) ;
 - * la vitesse d'échange de ces registres est de 8GB/s ;
 - ◊ Warp scheduler : ordonnancement des warps de 32 threads ;



512 «cuda cores» organisés en 16 SM de 32 cores chacun.

32 «cuda cores» par SM, «*Streaming Multiprocessor*».

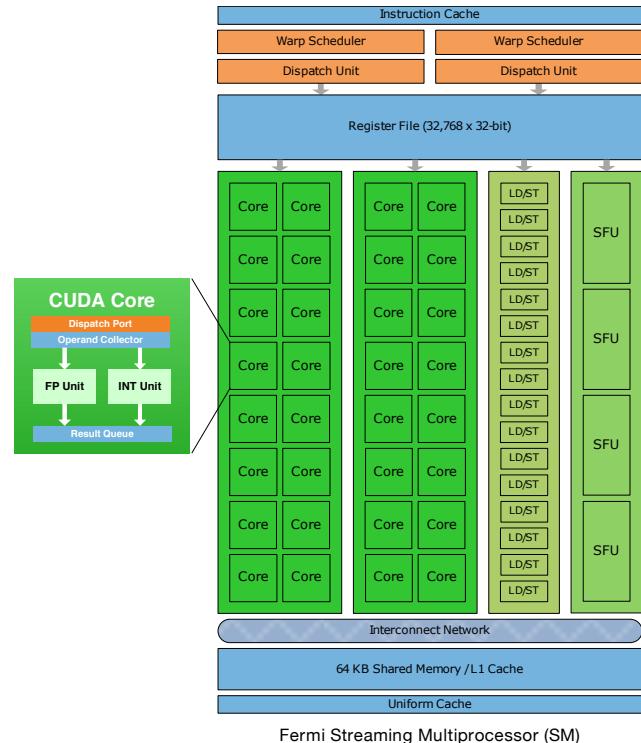
Un «cuda core» correspond à un circuit capable de réaliser un calcul en simple précision, FPU, ou sur un entier, ALU (opérations binaire comprises), en 1 cycle d'horloge.

Une unité «*Special Function Unit*» exécute les opérations transcendentales comme le *sin*, *cos*, *sqrt*, *arccos*, etc.

Il en existe 4 : seuls 4 opérations de ce type peuvent avoir lieu simultanément.

Le GPU dispose de 6 partition mémoire de 64 bits, soient 16 unités sur 32bits.

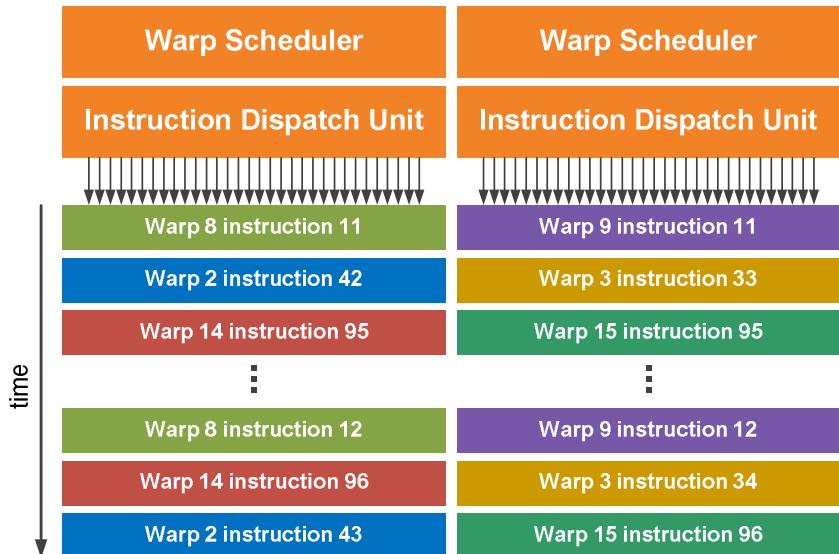
16 unités «*Load/Store*» permettent d'accéder en écriture ou en lecture à des données pour 16 threads par cycle d'horloge.



Deux «*Warp Schedulers*» permettent de programmer l'exécution de deux Warps indépendants simultanément :

- seuls les warps capables de s'exécuter peuvent être exécutés simultanément ;
- un scheduler programme l'exécution d'une instruction d'un Warp vers un groupe de 16 cores/16 LD/ST or 4 SFUs.
- la plupart des instructions peuvent être émises par deux : deux instructions entières, deux instructions flottantes, un mix d'entier de flottant, lecture, écriture et opération transcendentale.

Mais les instructions en double précision ne peuvent pas être émises par deux.



Des opérations spéciales

MAD, «*multiply-add*» : une opération de multiplication et d'addition combinée, mais avec perte des bits dépassant la capacité.

FMA, «*fused multiply-add*» : réalise les mêmes opérations mais sans perte, en simple ou double précision.

L'utilisation de calcul en double précision permet 16 calcul de type MAD par SM et par cycle d'horloge.

Multiply-Add (MAD):

$$\begin{array}{c} \boxed{A} \times \boxed{B} = \boxed{\text{Product}} \text{ (truncate extra digits)} \\ + \\ \boxed{C} = \boxed{\text{Result}} \end{array}$$

Fused Multiply-Add (FMA)

$$\begin{array}{c} \boxed{A} \times \boxed{B} = \boxed{\text{Product}} \text{ (retain all digits)} \\ + \\ \boxed{C} = \boxed{\text{Result}} \end{array}$$

La mémoire

- cache L1/mémoire partagée, utilisable au choix :
 - ◊ en mémoire partagée entre les threads pour leur permettre de coopérer en échangeant des données ;
 - ◊ pour conserver les valeurs des registres nécessaires à une thread qui ne peuvent être affectées à des registres disponibles dans le SM. *On parle de «spilled» registers*, c-à-d du débordement de registres ;
 - ◊ débit : 1600Go/s ;
 - ◊ organisable en 16Ko de cache L1 et 48Ko de mémoire partagée ou 48Ko de cache L1 ;
 - ◊ organisable en 48Ko de cache L1 et 16Ko de mémoire partagée ou 48Ko de cache L1 ;
- cache L2 : 768Ko partagé entre les 16 SMs qui sert aux :
 - ◊ accès vers/depuis la mémoire globale ;
 - ◊ copies depuis/vers le Host ;
 - ◊ accès aux textures.

