

# GPGPU<sub>s</sub>

Xavier JUVIGNY

ONERA

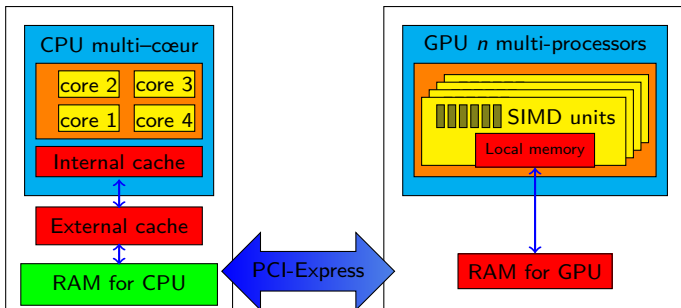
October 22, 2021

# Plan du cours

# Relation CPU-GPGPU

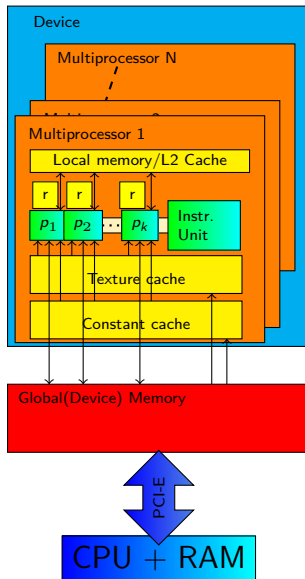
## Définition

- Le GPGPU est contrôlé par le CPU comme calculateur hybride MIMD-SIMD pour exécuter des algorithmes adaptés à son architecture;
- CPU et GPGPU sont des calculateurs multi-cœurs et ont une mémoire architecturée sous forme hiérarchique.



# Détail de l'architecture GPGPU

- GPGPU : Ensemble de  $N$  petites unités SIMD indépendantes partageant une mémoire globale commune :  $N$  multiprocesseurs;
- Multiprocesseur : Petite unité SIMD avec :
  - $k$  ALU synchronisés;
  - 1 décodeur d'instruction;
  - Trois mémoires partagées pour tous les ALUs (dont deux mémoires caches)
  - $R$  registres distribués parmi les ALUs (locales à chaque *thread*) (Exemple Maxwell : 65536)



# NVIDIA : système de numérotation hardware

## Numéros de version NVIDIA/Cuda

Deux systèmes de numérotation de version :

- 1 **Numérotation du hardware** : Un numéro majeur donnant l'architecture mise en œuvre sur le GPGPU utilisé, un numéro mineur donnant les améliorations qui ont pu y être apportées ( Exemple : parallélisme dynamique qu'à partir du hardware 3.5 ).
- 2 **Numérotation du driver** : La version de la bibliothèque Cuda utilisée ( 10.2 pour la plus récente ).

## Comment connaître ses numéros de version

- 1 Par l'application deviceQuery (voir prochains transparents );
- 2 En utilisant l'API C : `cudaGetDeviceProperties`

```
cudaDeviceProp prop;  
cudaGetDeviceProperties(&prop, num_device);  
std::cout << "Capabilité: " << prop.major << ". "  
            << prop.minor << std::endl;
```

# queryDevice

## Utilitaire queryDevice

- Fourni avec les “Samples” proposés à l'installation par NVIDIA ou téléchargeables à part;
- Doit être compilé avant utilisation !
- Localisé au niveau des Samples dans 1\_Uutilities/deviceQuery

## Exemple sortie obtenue ( vue partielle )

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 970M"

CUDA Driver Version / Runtime Version	8.0 / 8.0
CUDA Capability Major/Minor version number:	5.2
Total amount of global memory:	3040 MBytes (3187343360 bytes)
(10) Multiprocessors, (128) CUDA Cores/MP:	1280 CUDA Cores
GPU Max Clock rate:	1038 MHz (1.04 GHz)
Memory Clock rate:	2505 Mhz
Memory Bus Width:	192-bit
L2 Cache Size:	1572864 bytes
...	

# Organisation des cœurs de calcul

## Multiprocesseurs

- Un GPGPU contient plusieurs multi-processeurs ( 10 dans notre exemple );
- Chaque multi-processeur contient une mémoire locale, des registres et un nombre de cœur ( 128 dans notre exemple );
- Les cœurs de calcul sont organisés par groupe ( Warp ) de 16 ou 32 threads ( selon les architectures ).
- Un Warp est constitué de deux demi-warps. Un demi-warp possède une architecture SIMD.

# Organisation de la mémoire sur GPGPU

## Hiérarchie mémoire

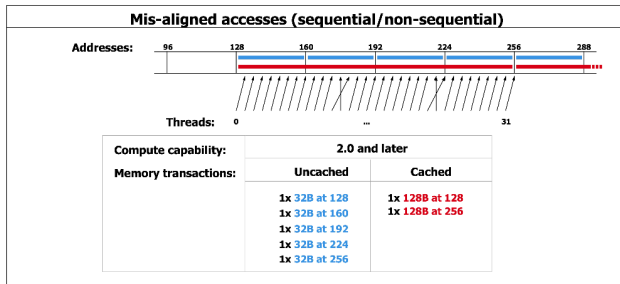
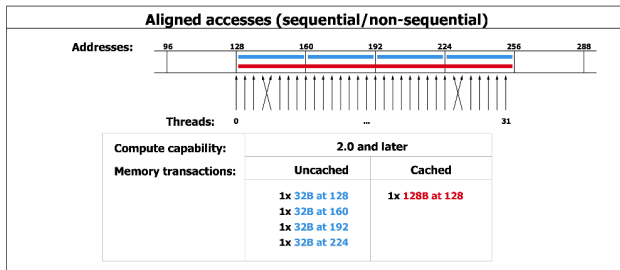
- 1 Chaque thread possède sa propre mémoire locale ( registres ), éventuellement partagée avec les threads appartenant au même Warp.
- 2 Chaque thread partage la même mémoire que les threads appartenant au même “multi–processeur”;
- 3 Tous les threads partagent la même mémoire globale;

## Coalescence

- 1 La mémoire globale est une mémoire entrelacée à 6 ou 12 voies ( dont deux de contrôle ) de largeur 32 octets;
- 2 Les threads d'un même warp accèdent à la mémoire globale par accès de 128 octets : une requête pour des données sur quatre octets, deux requêtes pour des données de huit octets, soit une requête par demi–warp, quatre octets pour des données de seize octets, soit une requête par quart de warp.
- 3 Pour cela, les données lues et écrites par un warp doivent être contiguës en mémoire et alignées sur 128 octets.



# Coalescence



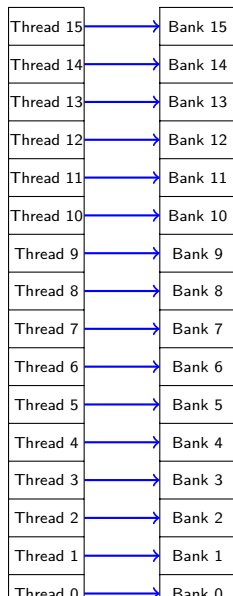
# Mémoire partagée

- Des centaines de fois plus rapide que la mémoire globale
  - 16 bancs peuvent être accédés simultanément sur un hardware 1.X
  - 32 bancs peuvent être accédés simultanément sur un hardware 2.0
  - 32 octets consécutifs sont assignés à des bancs successifs
- Des Threads d'un même bloc peuvent coopérer via la mémoire partagée
  - 16 KBytes maximum par multiprocesseur avec un hardware 1.X
  - 48 KBytes maximum par multiprocesseur avec un hardware 2.0
  - Mais sur le hardware 2.0, la mémoire cache L1 est la même mémoire que la mémoire partagée : le programmeur doit contrôler la taille de mémoire utilisée par le cache L1 et la mémoire partagée.
- Permet d'éviter des accès non coalescent en mémoire globale

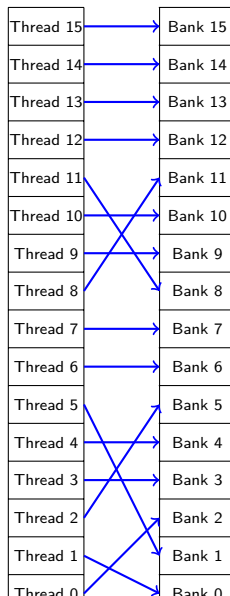
# Mémoire partagée : problèmes de performance

- Les cas idéaux :
  - Si tous les threads d'un demi-warp (ou un warp pour le hardware 2.0) accèdent à des bancs différents, pas de conflit de bancs
  - Si tous les threads d'un demi-warp (un warp en 2.0) lisent une adresse identique, pas de conflit de bancs (broadcast)
- Les pires cas :
  - Conflit de banc : Plusieurs threads d'un même (1/2)-warp accèdent à un même banc
  - L'accès est sérialisé
  - Coût =  $\max \#$  d'accès simultanés à un même banc

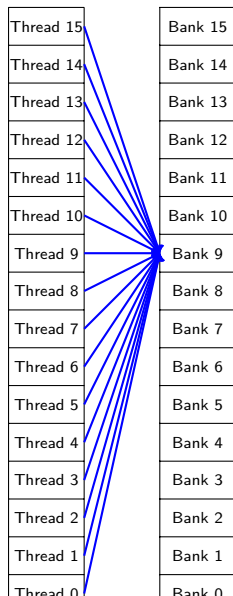
# Accès à la mémoire partagée



**Motif d'accès sans conflits de bancs :** chaque thread du demi-warp accède à un banc différent.

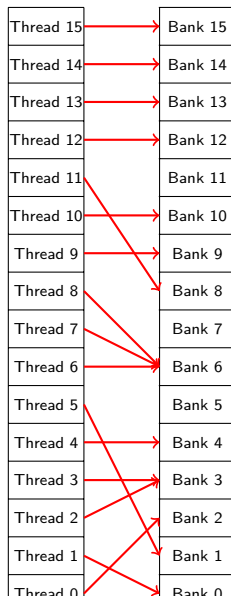


# Accès mémoire partagée



← Chaque thread lit une adresse d'un même banc : pas de conflit (broadcasting)

Plusieurs Threads accèdent au même banc : **conflit** →



# Principe de compilation CUDA et C++

Plusieurs cas de figure :

- Compilation d'un code entièrement développé en CUDA;
- Compilation d'un code CUDA avec récupération de code C/C++;
- Compilation code CUDA avec compilateur spécifique pour la partie C/C++ sur CPU.

# Compilation d'un code entièrement développé en CUDA

## Contenu et production du code

- Définitions variables et fonctions avec “qualificateurs” CUDA.
- Du code C ou C++ avec fonctionnalités CUDA;
- Code C ou C++ “standard”.
- Les extensions : “.h” pour les headers, “.cu” pour les sources.
- On compile à l'aide du compilateur NVidia : `nvcc`
- On obtient un code CPU contenant du code GPU intégré.

## Pour les codes C/C++ simples

- Possibilité de tout compiler avec `nvcc` dans des fichiers `.cu`
- Mais les optimisations pour le CPU peuvent en souffrir.

# Compilation d'un code avec récupération sources C/C++

## Contenu et production du code

- On compile les fichiers C/C++ (.c, .cc, .h) avec nvcc;
- Les fichiers contenant du code Cuda (.cu, .h) avec nvcc;
- On fait une édition des liens du tout pour obtenir un code binaire contenant les binaires pour le CPU et le GPU.

## Problèmes

- A l'édition des liens, des problèmes peuvent apparaître avec des templates...
- Problèmes d'optimisations pour le code CPU pouvant apparaître.



# Compilation d'applications CUDA avec compilateur spécifique

## Contenu et production du code

- Codes C/C++ (.c, .cc, .h) : On le compile avec son compilateur préféré (gcc, g++, icc, ...);
- Code Cuda : On le compile avec nvcc;
- On fait l'édition de lien des objets obtenus

## Problèmes

- Des problèmes de nommage peuvent apparaître (mais pas avec **gcc**).

# Principe d'exécution

## Exécution d'une application CUDA

- On lance une application CPU d'apparence classique;
- On réalise du “Remote Process Control” (RPC) sur le GPU depuis le CPU (exécution de “kernels”);
- Pour être efficace, il faut minimiser les transferts des données;
- On peut exécuter les “kernels” en mode bloquant (synchrone) ou non bloquant (asynchrone) pour le programme CPU : → possibilité d'utiliser simultanément le CPU et le GPU.

# C étendu

- **Nouv. déclarations** : global, device, shared, local, constant

```
__device__ float filter[N];  
__global__ void convolve(float* image) {  
    __shared__ float region[M];
```

- **nouveaux mots clefs** : threadIdx, blockIdx

```
    region[threadIdx] = image[i];
```

- **Intrinsics** : \_\_syncthreads

```
    __syncthreads(); image[j] = result;
```

- **API d'exécution** : Memory, symbol, execution management

```
void* mylmg = cudaMalloc(bytes); // Alloue memoire sur GPU
```

- **Exécution de fonction**

```
convolve<<<100,10>>>>(mylmg); // 100 blocs de 10 threads
```

# “Qualifieurs” de CUDA

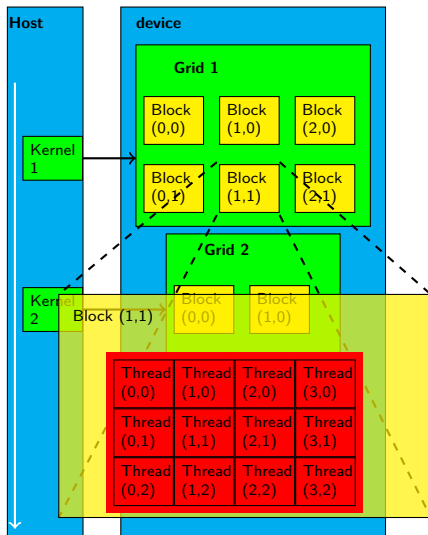
## Propriétés des “qualifieurs” de CUDA:

	<code>__device__</code>	<code>__host__</code>	<code>__global__</code>
Fonctions	Appel sur GPU Exécution sur GPU	Appel sur CPU Exécution sur CPU	Appel sur CPU Exécution sur GPU
	<code>__device__</code>	<code>__constant__</code>	<code>__shared__</code>
Variables	Mémoire globale GPU Temps de vie de l'application  Lisible/enregistrable sur CPU et GPU	Mémoire constante GPU Temps de vie de l'application  Enregistrable CPU, lisible GPU	Mémoire partagé multi-processeurs Temps de vie du bloc de thread  Lisible sur GPU : utilisé comme cache mémoire géré à la main pour la mémoire global GPU

→ Les qualifieurs séparent les codes CPU et GPU.

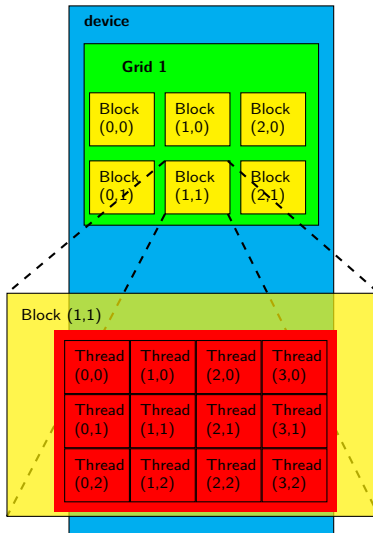
# Distribution des threads : grilles et blocs

- Un nœu est exécuté comme une grille de blocs de thread
  - Tous les threads partagent le même espace de mémoire de données
- Un bloc de threads est un ensemble de threads qui peuvent coopérer les uns les autres en :
  - synchronisant leur exécution
  - partageant leurs données à travers une mémoire partagée rapide
- Deux threads provenant de deux blocs différents ne peuvent pas coopérer :
  - Opérations atomiques



# Identification des blocs et des threads

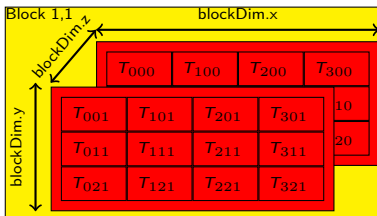
- Chaque thread et bloc ont des Ids :
  - Chaque thread peut décider sur quelles données travailler
  - Block ID : 1D, 2D ou 3D depuis Cuda 3.0
  - Thread ID: 1D, 2D ou 3D.
- Simplifie l'adressage mémoire quand on gère des données multidimensionnelles :
  - Image processing
  - Résolution d'EDP sur des volumes ou surfaces
  - ...



# Mots clés pour les blocs et les threads

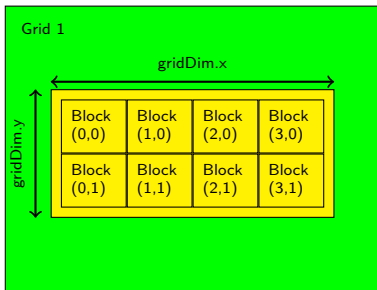
- Mots clés pour les blocs :

- `threadId.[x,y,z]` définit la position du thread dans le bloc;
- `blockDim.[x,y,z]` définit les dimensions du bloc.



- Mots clés pour les grilles :

- `blockId.[x,y,z]` définit la position du bloc dans la grille
- `gridDim.[x,y,z]` définit les dimensions de la grille



# Tableau de threads

Un noyau CUDA est exécuté par un tableau de threads

- Tous les threads exécutent le même code
- Chaque thread a un ID utilisé pour calculer les adresses mémoires et faire des contrôles pour le branchement (if, etc...)



```
float x = input[threadId];  
float y = func(x);  
output[threadId] = y;
```





# Thread ID

L'ID d'un thread dans un bloc est :

```
t_id = threadIdx.x +  
      threadIdx.y*(blockDim.x) +  
      threadIdx.z*(blockDim.x*blockDim.y);
```

- `threadIdx.[x,y,z]` : Indice du thread dans la dimension x,y,z
- `blockDim.[x,y,z]` : Taille du bloc dans la dimension x,y,z

# Thread ID(2)

- Considérons un bloc de dimension

```
blockDim.x = 8  
blockDim.y = 6  
blockDim.z = 4
```

- Et un thread d'indices

```
threadIdx.x = 1  
threadIdx.y = 2  
threadIdx.z = 3
```

- Le thread est alors d'indice global dans le bloc :

```
1+(2*8)+3*(6*8) = 161
```

# Exemple 1

## Addition de deux vecteurs

```
__global__ void addVector( int dim, const float* u,
                           const float* v, float* w )
{
    int ind = threadIdx.x + blockIdx.x * blockDim.x;

    if (ind < dim)
        w[ind] = u[ind]+v[ind];
}
```

## Exemple 2

### Addition de deux matrices

```
__global__ void addMatrix(float* A, float* B, float* C, int N)
{
    unsigned int iGlob = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int jGlob = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int ind    = iGlob + jGlob * N;
    if ((iGlob < N) && (jGlob < N)) C[ind] = A[ind] + B[ind];
}
```

## Exemple 3

Multiplication matrice-matrice :

```
#define BLOCK_SIZE 16
__global__ void
matrixMul( float* C, const float* A, const float* B, int dim )
{ // On fait une approche par bloc : 1 bloc pour un groupe de thread
  // Indice premier bloc lu par le thread
  int aBegin = dim * BLOCK_SIZE * blockIdx.y;
  int aEnd   = aBegin + dim - 1; // Et indice suivant dernier bloc
  int aStep  = BLOCK_SIZE; // Et pas pour prochain bloc

  int bBegin = BLOCK_SIZE * blockIdx.x; // indice 1er bloc
  int bStep  = BLOCK_SIZE * dim; // Pas pour prochain bloc

  // Chaque thread calcul un coefficient de C :
  int ic = dim * BLOCK_SIZE * blockIdx.y + BLOCK_SIZE * blockIdx.x;
  float Csub = C[ic + dim*threadIdx.y + threadIdx.x];
```

## Exemple 3 (suite)

### Multiplication matrice-matrice (suite):

```
...
// Boucle sur les blocs :
for ( int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep )
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Chaque thread du group charge un elt des blocs courants
    // de A et de B en shared memory :
    As[threadIdx.y][threadIdx.x] = A[a + dim*threadIdx.y + threadIdx.x];
    Bs[threadIdx.y][threadIdx.x] = B[b + dim*threadIdx.y + threadIdx.x];
    // On s'assure que tous les threads ont bien remplis As et Bs :
    __syncthreads();
    // Puis multiplication des deux blocs qu'on rajoute à Csub :
    for ( int k = 0; k < BLOCK_SIZE; ++k )
        Csub += As[threadIdx.y][k] * Bs[k][threadIdx.x];
    __syncthreads(); // On s'assure d'avoir fini le calcul bloc
}
C[ic + dim*threadIdx.y + threadIdx.x] = Csub;
}
```

# Caractéristiques de CUDA : facile et léger

- L'API est une extension du langage C → apprentissage aisé;
- Le hardware est conçu pour une exécution et une gestion des tâches légère → performance élevée.

# Allocation mémoire

- `cudaMalloc()`
  - Alloue des objets sur la **mémoire globale** du GPU
  - Deux paramètres nécessaires :
    - 1 Adresse du pointeur sur l'objet alloué;
    - 2 Taille de l'objet alloué;
- `cudaFree()`
  - Libère des objets de la mémoire globale du GPU;
    - 1 Pointeur sur l'objet à libérer;

Ex.: Alloue une matrice 1024\*1024 en simple précision

```
#define MATRIX_SIZE 1024*1024
float* MyMatrixOnDevice;
int size = MATRIX_SIZE*sizeof(float);
cudaMalloc((void*)&MyMatrixOnDevice, size);
cudaFree(MyMatrixOnDevice);
```



# Transfert de données en CUDA entre le CPU et le GPU

## cudaMemcpy()

- Transfert de données
- Quatre paramètres nécessaires :
  - Pointeur vers la source
  - Pointeur vers la destination
  - Nombre d'octets à copier
  - Type de transfert :
    - CPU vers CPU
    - CPU vers GPU
    - GPU vers CPU
    - GPU vers GPU

Des variantes asynchrones supportées depuis la version hardware 1.1HW

# Exemples de transfert CUDA entre le CPU et le GPU

- Exemple de code :

- Transfert une matrice 1024\*1024 en simple précision
- `MyMatrixOnHost` est un pointeur sur la mémoire du CPU et `MyMatrixOnDevice` est un pointeur sur la mémoire globale du GPU
- `cudaMemcpyHostToDevice` et `cudaMemcpyDeviceToHost` sont des constantes symboliques

```
cudaMemcpy( MyMatrixOnDevice , MyMatrixOnHost , size ,  
            cudaMemcpyHostToDevice );  
cudaMemcpy( MyMatrixOnHost , MyMatrixOnDevice , size ,  
            cudaMemcpyDeviceToHost );
```

# Déclaration de fonctions CUDA

	Exécuté sur	Appelable seulement de
<code>__device__ float DeviceFunc()</code>	GPU	GPU
<code>__global__ void KernelFunc()</code>	GPU	CPU
<code>__host__ float HostFunc()</code>	host	host

- `__global__` définit une fonction noyau : doit retourner toujours `void`.
- `__device__` fonctions sur GPU dont on ne peut récupérer l'adresse (semblable à des fonctions inline);
- Pour les fonctions exécutées sur le GPU :
  - Pas de fonctions récursives
  - Pas de déclaration de variables statiques dans la fonction
  - Pas de nombre d'arguments variables

# Appeler un noyau : création de threads

- Une fonction noyau doit être appelée avec une configuration d'exécution :

```
__global__ void KernelFunc (...);  
dim3 DimGrid(100,50); // 5000 Thread blocks  
dim3 DimBlock(8,8,4); // 256 threads per block  
  
KernelFunc<<<DimGrid, DimBlock>>>(...);
```

- Tout appel à un noyau est asynchrone, une synchronisation explicite nécessaire pour des rendez-vous.

# Optimiser le nombre de threads par bloc

- Choisir le nombre de threads par bloc comme un multiple de la taille d'un warp
  - Essayer d'éviter le gâchis de warp en sous effectifs
- Plusieurs threads par bloc = meilleur recouvrement de la latence mémoire
  - L'invocation de noyau peut se planter si trop de registres utilisés.
- Heuristiques
  - Minimum requis par le hardware : 64 Threads par bloc
    - Seulement si beaucoup de blocs concurrents
  - 192 ou 256 threads est un meilleur choix :
    - Généralement assez de registre pour arriver à compiler et exécuter
  - Tout cela dépend de votre calcul, alors expérimentez !

# Heuristique taille Grille/Bloc

- $\# \text{ de blocs} > \# \text{ de multiprocesseurs}$ 
  - Pour que tous les multiprocesseurs aient au moins un bloc à exécuter
- $\# \text{ de blocs} / \# \text{ de multiprocesseurs} > 2$ 
  - Plusieurs blocs peuvent être en concurrence dans un multiprocesseur
  - Les blocs qui n'attendent pas un `__syncthreads()` sont toujours actifs
  - Selon les ressources valables – registre, mémoire partagée
- $\# \text{ de blocs} > 100$  pour s'adapter aux futurs hardware
  - Blocs sont exécutés en pipeline sur un multiprocesseur
  - 1000 blocs par grille devrait s'adapter aux générations futures de GPU

# Occupation

- Les instructions dans les threads sont exécutées simultanément, alors exécuter d'autres warps est le seul moyen de cacher les latences et de garder le hardware occupé.
- **Occupation** = nombre de warps s'exécutant en concurrence sur un multiprocesseur divisé par le nombre maximal de warps qui peut être exécuté en concurrence.
- Limité par l'utilisation des ressources :
  - Registres
  - Mémoire partagée
  - threads/blocs

# Cas d'occupation

## ● Hardware 1.0/1.1

768 threads :	$3 \times 256(16 \times 16)$	$8 \times 64$ (66% utilisé)
16 kBytes partagé	$3 \times 5\text{kbytes}$	$8 \times 1.9\text{kbytes}$
8192 registers	10 per thread	15 per thread
8 blocks	3 blocks	8 blocks

## ● Hardware 1.2/1.3

1024 threads :	$4 \times 256(16 \times 16)$	$8 \times 64$ (50% utilisé)
16 kBytes partagé	$4 \times 3.9\text{kbytes}$	$8 \times 1.9\text{kbytes}$
16384 registers	15 per thread	30 per thread
8 blocks	4 blocks	8 blocks

## ● Hardware 2.0

1024 threads :	$4 \times 256(16 \times 16)$	$8 \times 64$ (50% utilisé)
32 kBytes partagé	$4 \times 7.8\text{kbytes}$	$8 \times 3.8\text{kbytes}$
32768 registers	30 per thread	60 per thread
8 blocks	4 blocks	8 blocks



# Retour exemple 1

## Addition deux vecteurs : fonction appel noyau

```
void add_vector(const float* u, const float* v, float* w, int N)
{
    int grdSize, blockSize = 256;
    float *u_dev, *v_dev, *w_dev;
    // Alloue et copie les vecteurs u, v et alloue w sur le GPU
    cudaMalloc(&u_dev, sizeof(float)*N);
    cudaMemcpy(u_dev, u, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMalloc(&v_dev, sizeof(float)*N);
    cudaMemcpy(v_dev, v, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMalloc(&w_dev, sizeof(float)*N);
    // Calcule la configuration d'exécution du noyau
    dim3 dimBlock(blockSize);
    grdSize = (N%blockSize>0 ? N/blockSize+1: N/blockSize);
    dim3 dimGrid(grdSize);
    // Appel du noyau :
    addVector<<<dimGrid, dimBlock>>>>(N, u_dev, v_dev, w_dev);
    // Copie le resultat sur le CPU et libere la memoire GPU
    cudaMemcpy(w, w_dev, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(u_dev); cudaFree(v_dev); cudaFree(w_dev);
}
```