Institut
Mines-Télécom

Architecture et Programmation GPU

Elisabeth Brunet

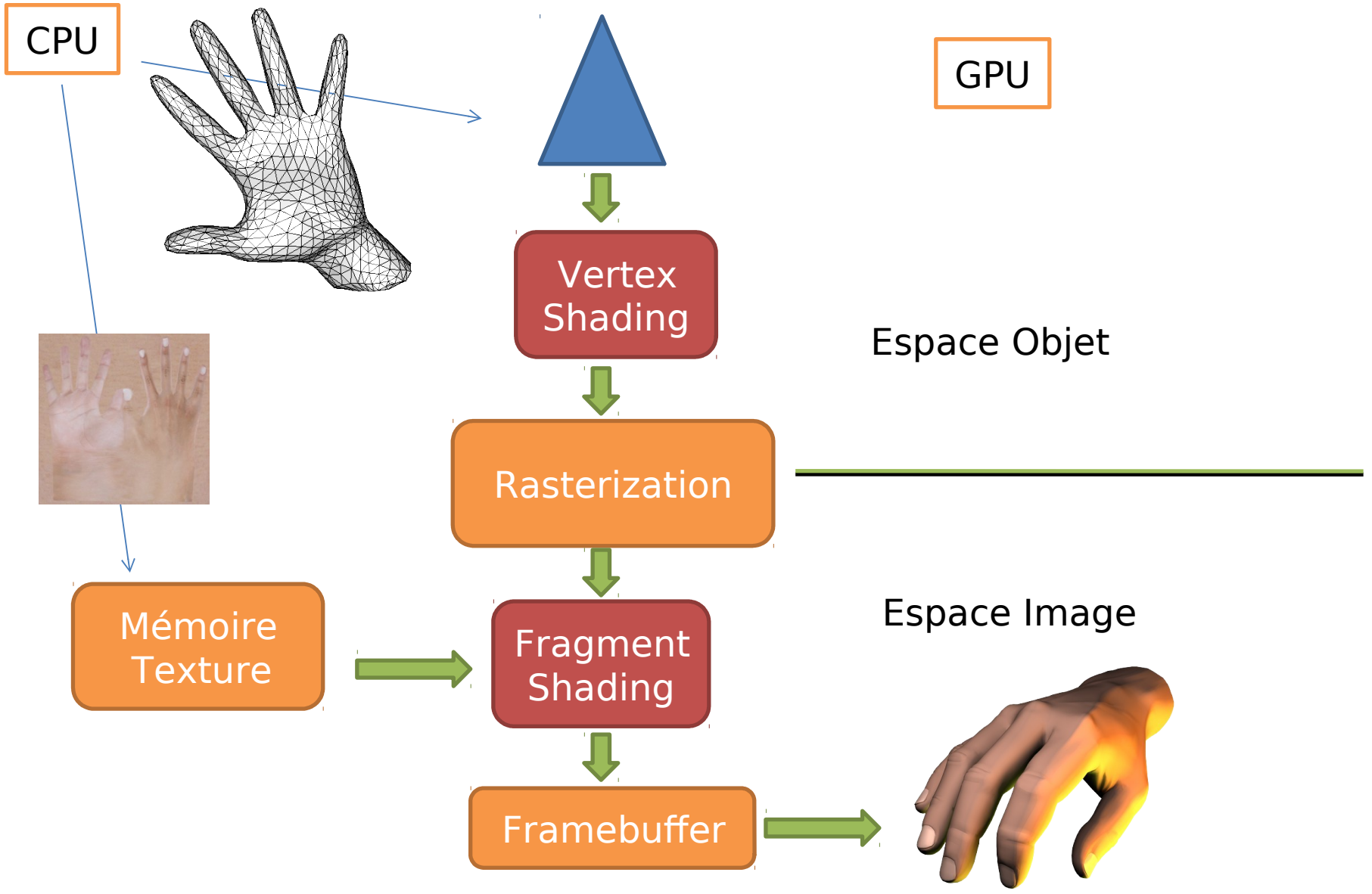
Plan

- Introduction
- Architecture GPU
- Programmation CUDA
- Conclusion

GPU

- Graphics Processor Unit
- Composant matériel indépendant
  - Co-processeur sur la carte mère sur un slot pci-express
  - Architecture many-coeur + mémoire propre
- Initialement,
  - Conçu pour le calcul 3D en synthèse d’image
  - Poussé par le marché du jeu vidéo

Pipeline graphique

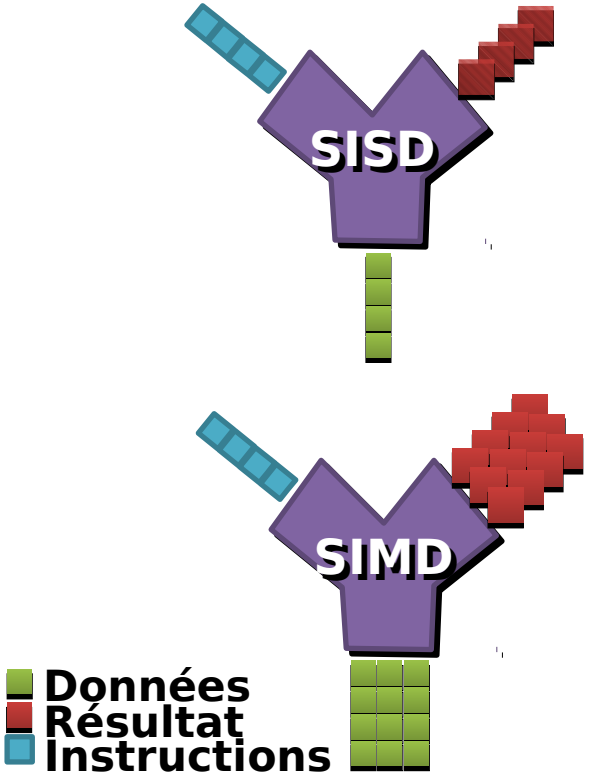


GPU

- Graphics Processor Unit
- Composant matériel indépendant
  - Co-processeur sur la carte mère sur un slot pci-express
  - Architecture many-coeur + mémoire propre
- Initialement,
  - Conçu pour le calcul 3D en synthèse d'image
  - Poussé par le marché du jeu vidéo
- Depuis ~2000, GPGPU (pour General Processing GPU)
  - Ouverture de l'architecture
  - Opérations de calcul plus générales
    - Traitement d'images, simulations numériques, algèbre linéaire, deep learning, etc.

GPU

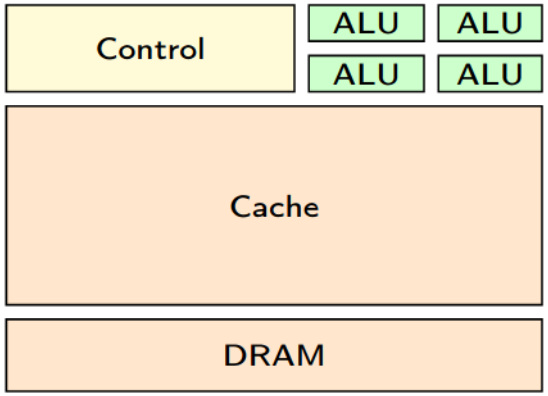
- Architecture **SIMD**
  - **Single Instruction / Multiple Data**
  - Parallélisme de données
  - Idéal pour le calcul intensif massivement parallèle
- Centaines de cœurs
- Cœurs aux capacités limitées
  - Pas d'allocation dynamique de mémoire
  - Pas de pile > pas de récursion
- Hiérarchie de mémoire
  - Effets NUMA



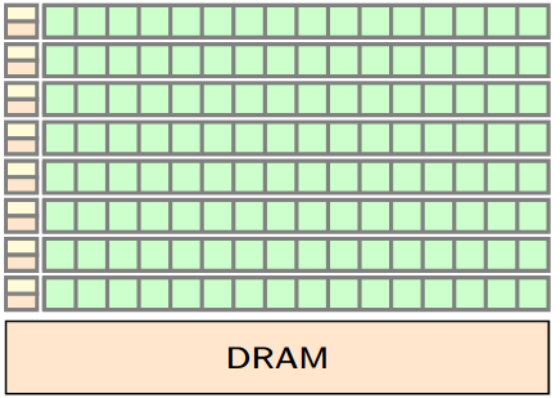
CPU vs GPU

- Latence
  - Délai entre l'initialisation d'une opération et le moment où ses effets sont détectables
  - Une voiture a une latence plus faible qu'un bus
- Bande passante
  - Quantité de travail atteint sur une durée donnée
  - Un bus a une bande passante plus élevée qu'une voiture

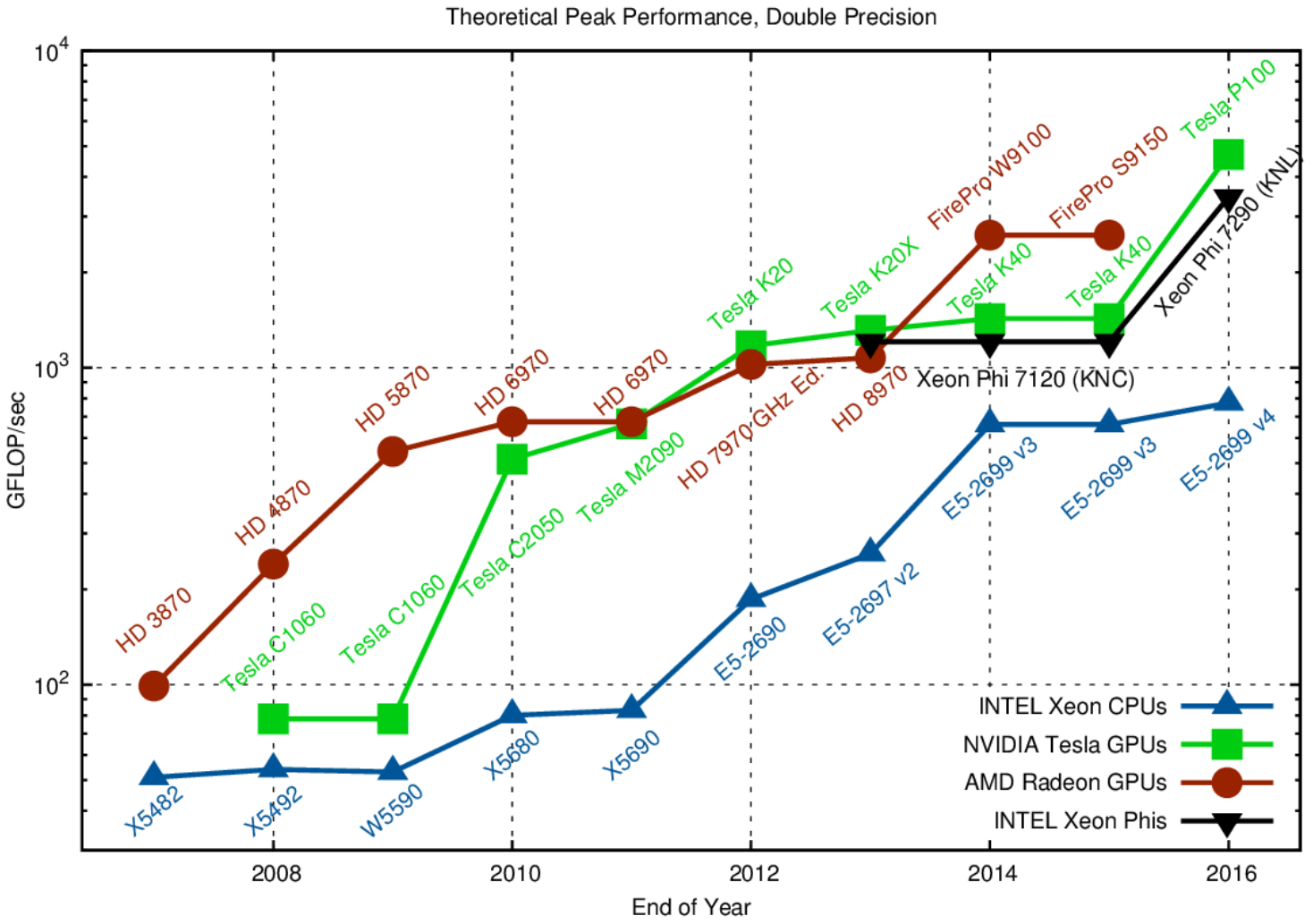
- CPU
  - Architecture qui vise à minimiser la latence
    - Opérations exigeantes, e.g. événement clavier
    - En s'appuyant sur les caches
    - Circuits dédiés pour les opérations en dehors du cache
      - e.g. pre-fetch, exécution out-of-order



- GPU
  - Processeurs à latence et bande passante élevées
    - Pas besoin de large cache
    - → Transistors dédiés au traitement des données plutôt qu'à la gestion des caches
    - → Chip de même taille mais avec beaucoup plus d'ALU

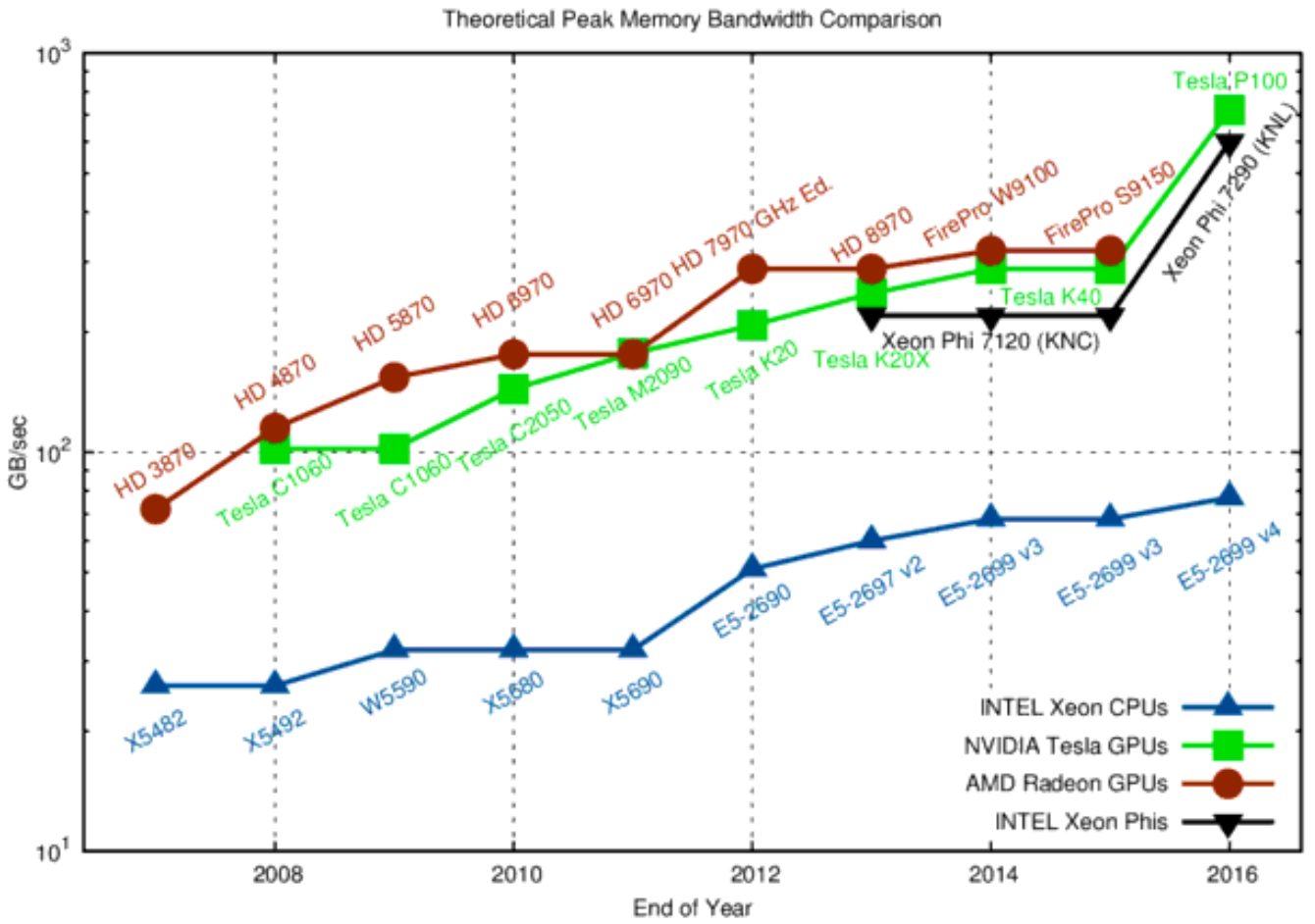


CPU vs GPU



TELECOM SudParis

CPU vs GPU


Theoretical Peak Memory Bandwidth Comparison
GB/sec
End of Year
2008
2010
2012
2014
2016
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Tesla P100
Xeon Phi 7290 (KNL)
Tesla K40
Xeon Phi 7120 (KNC)
Tesla C1060
Tesla C1060
Tesla C2050
Tesla M2090
Tesla K20
Tesla K20X
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis

GPUs concurrents

- Cartes GeForce/Quadro/Tesla d'NVIDIA
  - Micro-architectures Fermi, Kepler, Maxwell, Pascal, Volta, Turing, Ampere
  - Programmation orientée calcul : CUDA
- Cartes Radeon d'AMD
  - Incluant les architectures Stream Computing d'ATI
  - OpenCL : normalisation de la programmation
- Programmation graphique : OpenGL, Vulkan, Direct3D, DirectX

Architecture Turing

INTRODUCING TURING

TU102 – FULL CONFIG

18.6 BILLION TRANSISTORS

SM	72
CUDA CORES	4608
TENSOR CORES	576
RT CORES	72
GEOMETRY UNITS	36
TEXTURE UNITS	288
ROP UNITS	96
MEMORY	384-bit 7 GHz GDDR6
NVLINK CHANNELS	2

THIS PRESENTATION IS EMBARGOED UNTIL SEPTEMBER 14, 2018

Tensor cores

- Cœurs spécialisés
  - 4x4 matrix cores
  - Ultra rapides pour les opérations sur toutes petites matrices
    - 1 matrix multiply-accumulate operation per 1 GPU clock
  - Particulièrement adaptés aux demandes du deep learning
  - https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

ARCHITECTURE CUDA

Description

- *Compute Unified Device Architecture*
- Architecture matérielle et logicielle des GPU Nvidia
- Programmable en C, C++, Fortran, Python
- Exploite directement l’architecture unifiée (G80 et +)

Principe

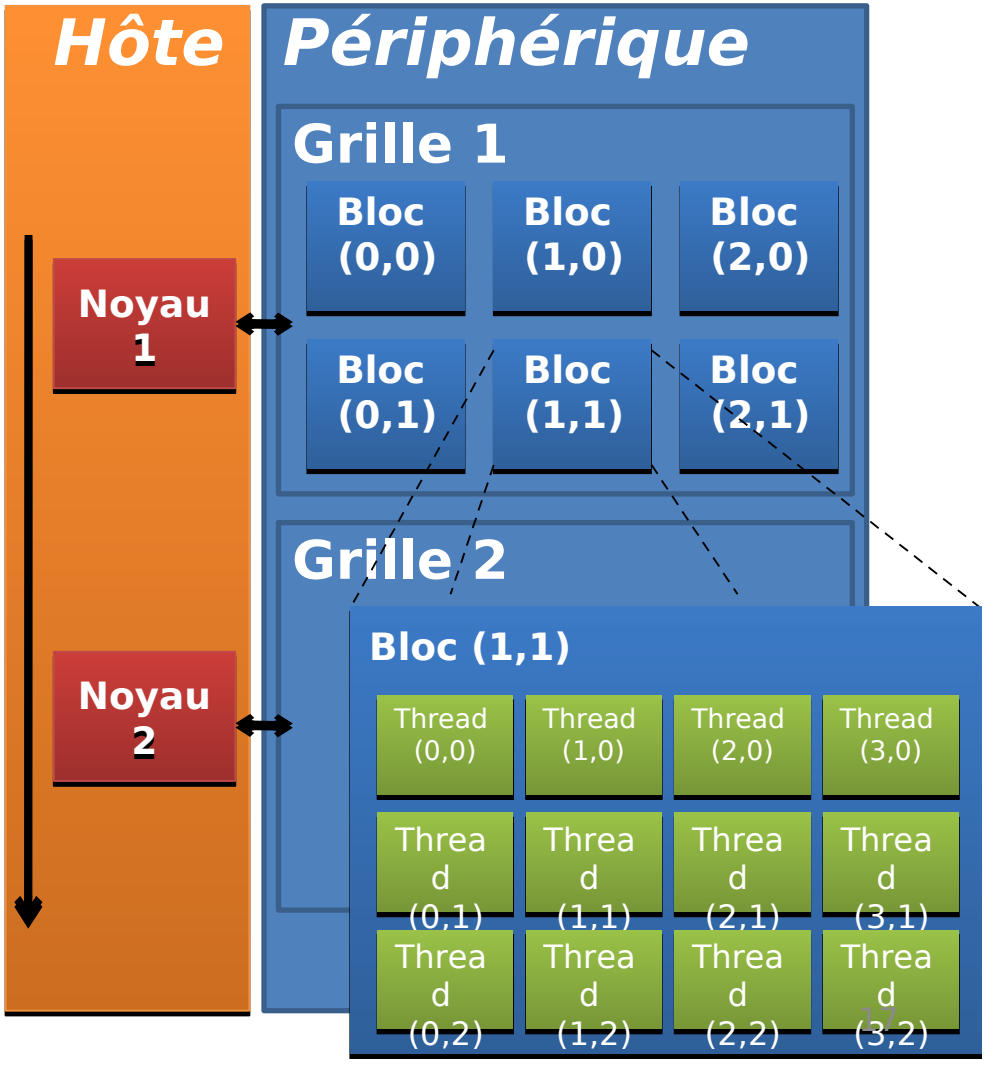
- Depuis un programme hôte lancé sur un CPU (*host*),
  - Lancement d'un noyau de calcul (*kernel*) sur le GPU(*device*) qui
    - Exécute le même calcul
      - Grâce à de nombreux threads très légers
    - Sur différentes données chargées dans la mémoire du GPU
  - → Calcul déporté

  - Depuis Fermi, plusieurs noyaux lancés en parallèle
  - Depuis Kepler, lancement de noyaux depuis un noyau

  - Pas de gestion explicite de l'ordonnancement des threads, politique mémoire fixe

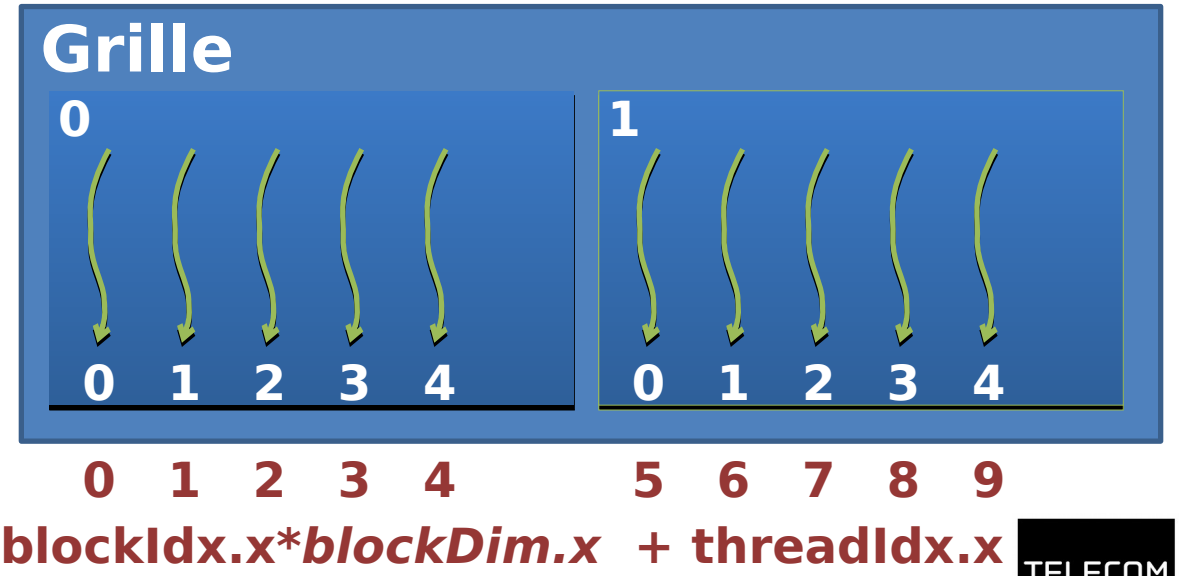
Modèle de programmation

- Noyau exécuté par une grille de blocs de threads
  - Grille 2D
  - Bloc 3D
- Dans un bloc, les threads
  - Coopèrent via de la mémoire partagée
  - Sont ordonnancés par *warp*
    - *Warp = 32 threads*
  - Threads d'un warp synchrones
- Pas de coopération inter-bloc

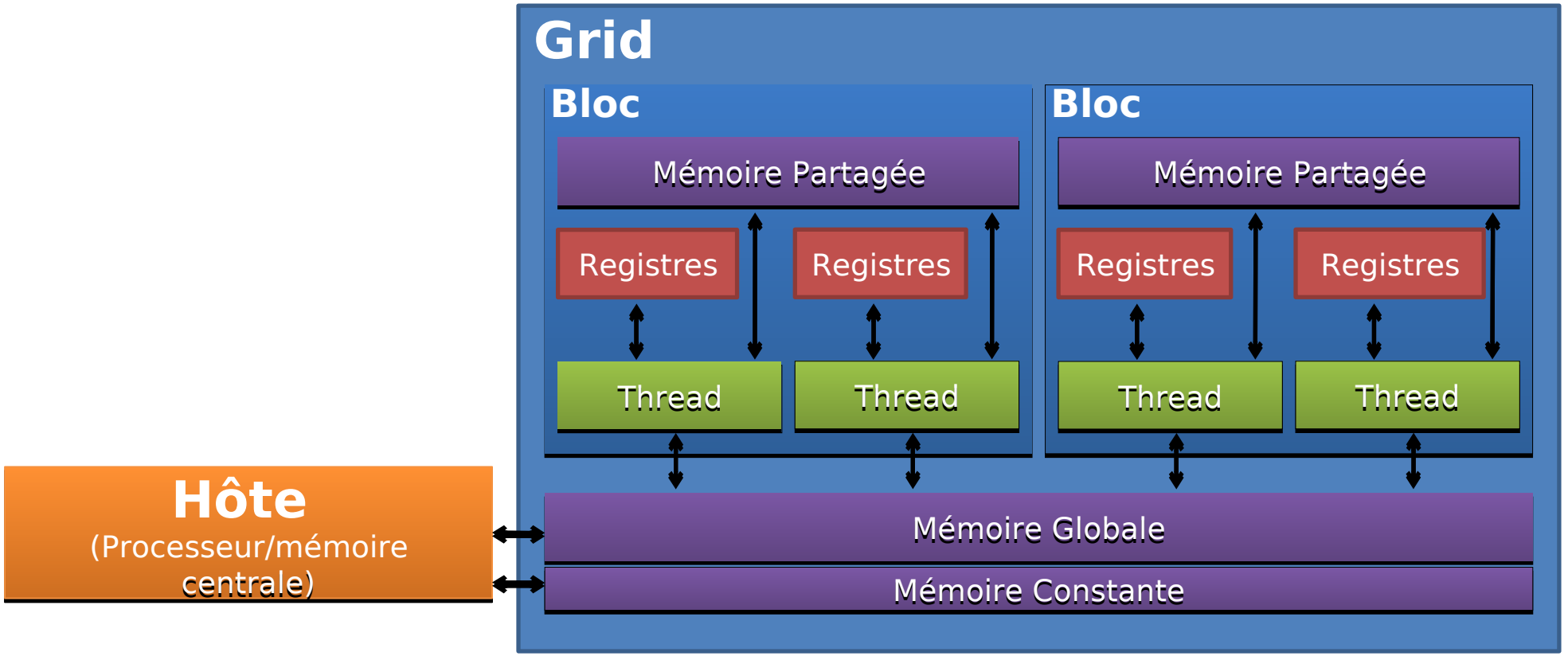


Identifiants multi-dimensionnels

- Chaque thread accède à une partie différente des donnés
  - Se conforme à la structure des données
- Informations d'indexation d'un thread
  - threadIdx.x, .y, .z : index du thread au sein du bloc
  - blockIdx.x, .y : index du bloc au sein de la grille
- Information sur la grille de threads exécutée
  - blockDim.x, .y, .z
  - GridDim.x, .y



Mémoire


Grid
Bloc
Bloc
Mémoire Partagée
Mémoire Partagée
Registres
Registres
Registres
Registres
Thread
Thread
Thread
Thread
Hôte
(Processeur/mémoire centrale)
Mémoire Globale
Mémoire Constante

PROGRAMMATION CUDA C

Informations sur le GPU

- Pour adapter l'exécution aux capacités de calcul
- Nombre de GPUs disponibles : **cudaGetDeviceCount**
- Caractéristiques d'un GPU : **cudaGetDeviceProperties**
  - Modèle, **t**aille de la mémoire, etc.

- Choix du GPU

  à partir de critères :

  cudaChooseDevice

```
cudaDeviceProp  prop ;
int  count;

cudaGetDeviceCount(&count) ;

for(int i = 0;  i < count ;  i++){
  cudaGetDeviceProperties (&prop ,  i) ;
  printf («[GPU%d] Taille de la mémoire globale%ld\n »,
              i, prop.TotalGlobalMem ) ;
}
```

Noyau de calcul CUDA

- __global__ void mon_noyau(*paramètres){...}*

- Un noyau est une fonction C avec quelques restrictions
  - Identifié grâce au mot clé __global__
  - Invoqué par le CPU et s'exécute sur le GPU
  - N'accède qu'à la mémoire GPU
  - Retour vide (void)
  - Pas de nombre variable d'arguments
  - Pas de récursion
  - Pas de variable statique
  - Les argument des noyaux sont copiés du CPU au GPU automatiquement
  - Les instructions de contrôle (if, while, for, switch, do)
    - Sérialisation des branches (au sein d'un warp)→ baisse de performance

Invocation de noyau

- mon_noyau <<<dim3 Grille,dim3 Bloc>>>(paramètres)

- Nombre maximum de threads par bloc : 1024 à répartir sur les 3 dimensions
- Dimension maximale de la grille : 2^31-1 x 65535 x 65535
- ➔ Informations liées à la spécification du GPU utilisé

- Variables prédéfinies paramétrées par l'invocation
  - dim3 gridDim : dimensions de la grille
    - Attention ! 2D au plus
  - dim3 blockDim : dimensions d'un bloc de threads

  - dim3 blockIdx : index du bloc dans la grille
  - dim3 threadIdx : index du thread dans le bloc

Exemple : Incrémenter un tableau

Code CPU

```
void incr_cpu(float *a,  float b, int N){
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main(){
    .....
    incr_cpu(a, b, N);
}
```

Code GPU

```
__global__ void incr_gpu(float *a, float b, int N){
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

void main(){
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    incr_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Fonctions

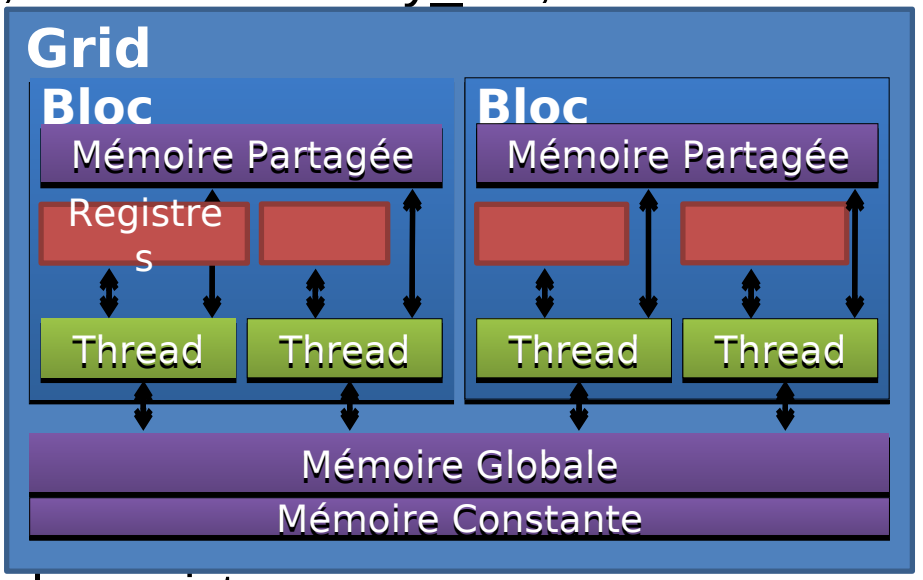
Qualificatif	Effet
__global__	Noyau de calcul Fonction invoquée depuis le code CPU et exécutée sur le GPU
__device__	Fonction invoquée et exécutée sur le device
__host__	(optionnel) Fonction invoquée et exécutée sur l'hôte

- __host__ et __device__ peuvent être combinés :
  - opérateurs surchargés
  - le compilateur génère 2 codes : une version CPU et une GPU

Différentes mémoires du GPU

- CPU et GPU ont des espaces mémoire physiquement séparés
- Sur le GPU :

1. Mémoire globale __device__
2. Mémoire partagée [__device__] __shared__
3. Mémoire constante [__device__] __constant__
4. Mémoire de texture : texture <Type, Dim, ReadMode> my_tex;



5. Registres : int localVar;
   - Sans qualification,
     - Allocation des variables scalaires dans les registres
     - Allocation des tableaux de plus de 4 éléments dans la mémoire globale

1. Mémoire globale

- Importante, forte latence, pas de cache
- Données
  - Accessibles par tous les threads de la grille
  - Durée de vie : en fonction des besoins
- Pilotée depuis l'hôte
  - Allocation/libération + mouvements de données
- Déclaration statique depuis le GPU avec __device__

1.Mémoire globale : Pilotage depuis le CPU

- Allocation : **cudaMalloc**(void ** pointer, size_t nbytes)
- Libération : **cudaFree**(void* p)
- Nettoyage : **cudaMemset**(void * p, int val, size_t nbytes)

```
// Allocation d'un tableau de n entiers
int n = 1024;
int nbytes = n*sizeof(int);
int *d_tab = NULL;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
...
cudaFree(d_a);
```

1.Mémoire globale : Pilotage depuis le CPU

- Mouvements de données depuis le CPU :
  cudaMemcpy(void *dst, void *src,
  size_t nbytes,
  enum cudaMemcpyKind direction);

  avec enum **cudaMemcpyKind**
  ={*cudaMemcpyHostToDevice,*
  cudaMemcpyDeviceToHost,
  cudaMemcpyDeviceToDevice}

- Copie une fois les appels CUDA précédents terminés
- Bloque le thread maître le temps de la copie

Exemple : Incrémenter un tableau

Code CPU

```
void incr_cpu(float *a,  float b, int N){
  for (int idx = 0; idx<N; idx++)
     a[idx] = a[idx] + b;
}


void main(){
  ...
  float*a=malloc(N*sizeof(float)) ;
  // initialisation de a
  incr_cpu(a, b, N);
}
```

Code GPU

```
__global__ void incr_gpu(float *a, float b, int N){
     int idx = blockIdx.x*blockDim.x + threadIdx.x;
     if (idx < N)    a[idx] = a[idx] + b;
}
void main(){
  ...
  float *a=malloc(N*sizeof(float)) ;
  // initialisation de a
  float *d_a = NULL ;
  cudaMalloc( (void**)&d_a, N*sizeof(float) );
  cudaMemcpy(d_a, a,N*sizeof(float),
                    cudaMemcpyHostToDevice) ;
   ...
  dim3 dimBlock (blocksize);
  dim3 dimGrid( ceil( N / (float)blocksize) );
  incr_gpu<<<dimGrid, dimBlock>>>(d_a, b, N);
  cudaMemcpy(a, d_a, N*sizeof(float),
     cudaMemcpyDeviceToHost) ;
}
```

2. Mémoire partagée

- Mémoire partagée __shared__
  - Espace séparé à très faible latence

```
// cas a
__global__void myKernel(){
        __shared__int shared[32];
        ...
}
```

- Données
  - Accessibles par tous les threads d'un même bloc
  - Durée de vie : exécution du noyau

- Allocation statique
  - Depuis le GPU
  - Taille statique donnée

    à la compilation (cas a)

    ou en début d'exécution du noyau (cas b)

```
// cas b
__global__void myKernel(){
        extern __shared__int s[];
        ...
}
int main() {
int size= numThreadsPerBlock* sizeof(int);
myKernel<<< dimGrid, dimBlock, size>>>();}
```

Synchronisation CPU

- Les noyaux de calcul sont asynchrones
  - Les appels CPU retournent tout de suite
  - Les noyaux s'exécutent après que tous les précédents se soient exécutés

- cudaMemcpy() est synchrone
  - L'appel CPU retourne après la réalisation de la copie
  - La copie démarre après que tous les appels CUDA précédents aient été exécutés

- **cudaThreadSynchronize()**
  - Bloque jusqu'à ce que tous les appels CUDA précédents se soient exécutés complètement

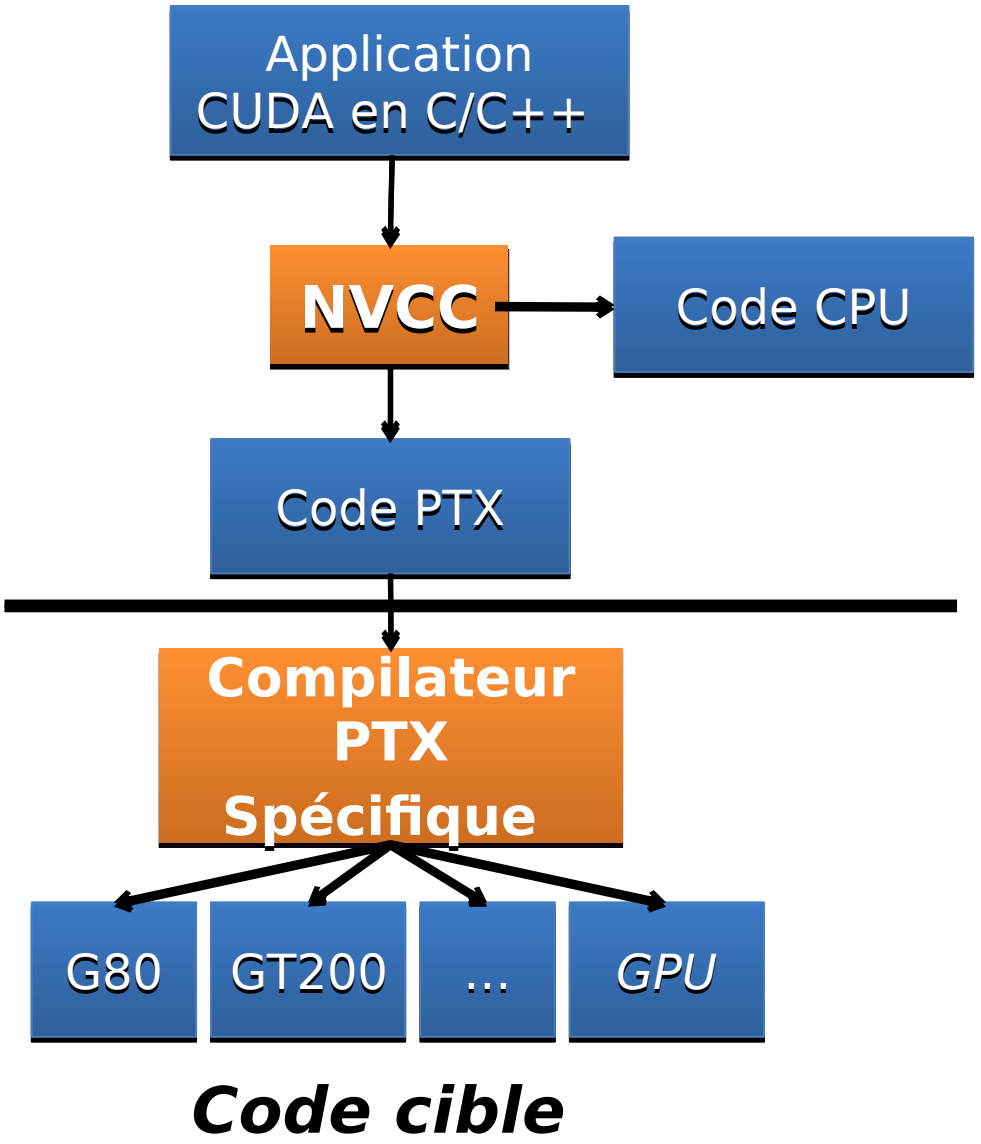
Synchronisation GPU

- void **__syncthreads();**
  - Synchronise tous les threads d'un bloc
    - *Barrière* de synchronisation
    - Autorisé sur code conditionnel seulement s'il est uniforme sur tout le bloc (ne dépend pas des Idx)

Gestion des accès concurrents (*race condition*)

- Opérations atomiques
  - Opérations non-interruptibles
    - atomicAdd()
    - atomicSub()
    - atomicMin()
    - atomicMax()
    - atomicInc()
    - atomicDec()
    - atomicExch()
    - atomicCAS()
    - atomicAnd()
    - atomicOr()
    - atomicXor()
- Outils d'exclusion mutuelle (mutex) non disponibles nativement
  - Implémentation proposée à base de variables partagées et d'atomicCAS (=comparaison d'une variable à une valeur donnée)
  - Problème d'exécution au sein des warps

Compilation



- Meta-compilateur nvcc
  - Code CPU et code GPU
- Exécutable contenant du code CUDA nécessite
  - CUDA core library (cuda)
  - CUDA runtime library (cudart) si utilisée

Gestion des erreurs d'exécution

- Tous les appels CUDA retournent un code d'erreur
  - Sauf pour les lancements de noyaux
  - **cudaError_t** type

- cudaError_t **cudaGetLastError**(void)
  - Retourne le retour d'exécution du dernier appel fait à CUDA
  - → Pour récupérer l'erreur d'exécution d'un noyau

- char* **cudaGetErrorString**(cudaError_t code)
  - Retourne une chaîne de caractères décrivant l'erreur
  - printf ("%s\n", cudaGetErrorString(cudaGetLastError ()));

Mesure de temps (1)
Avec un timer quelconque

- Les transferts de données sont synchrones

```
cudaMemcpy (d_x,  x, size, cudaMemcpyHostToDevice ) ;



kernel <<<g ,b>>>(d_x) ;



cudaMemcpy ( x ,  d_x , size, cudaMemcpyDeviceToHost) ;
```

Mesure de temps (1)
Avec un timer quelconque

- Les transferts de données sont synchrones
- L'appel au kernel ne l'est pas !!!

```
cudaMemcpy (d_x,  x, size, cudaMemcpyHostToDevice ) ;


t1 = myCPUTimer () ;

kernel <<<g ,b>>>(d_x) ;

cudaDeviceSynchronize () ;

t2 = myCPUTimer () ;


cudaMemcpy ( x ,  d_x , size, cudaMemcpyDeviceToHost) ;
```

Mesure de temps (2)
Avec un timer CUDA

- API CUDA event
- Les opérations sont séquentielles sur le GPU

```
cudaEvent_t  start,  stop ;

float milliseconds = 0.0;

cudaEventCreate(&start ) ; cudaEventCreate(&stop ) ;

. . .

cudaEventRecord(start) ;

saxpy <<<(N+255) /256 , 256>>>(N,  2.0 f ,  d_x ,d_y) ;

cudaEventRecord(stop) ;

cudaEventSynchronize(stop) ; //Garantit  que  l'événement s'est exécuté

cudaEventElapsedTime(&milliseconds, start, stop)
```

En route vers le TP !