GPU

Architecture et Programmation

Elisabeth Brunet et Tamy Boubekeur

Sommaire

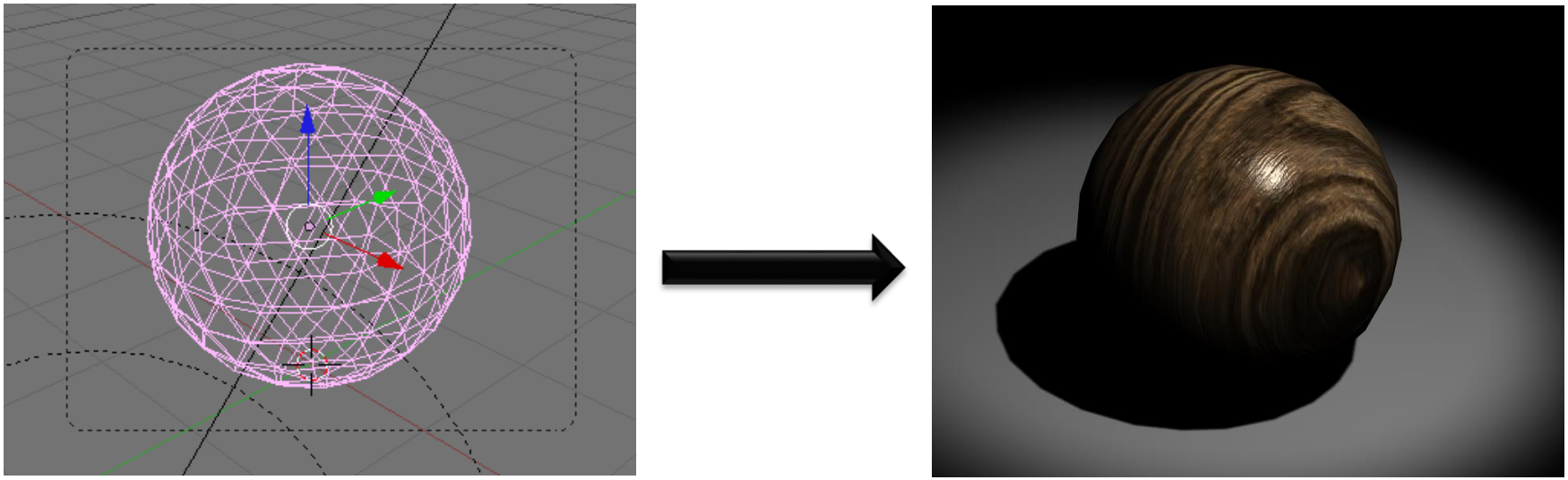
- Introduction
- Architecture CUDA
- Programmation CUDA
- Compilation CUDA
- OpenCL
- GPGPU Multi-passes vs CUDA/OpenCL
- Etudes de Cas
- Et après

INTRODUCTION

GPU

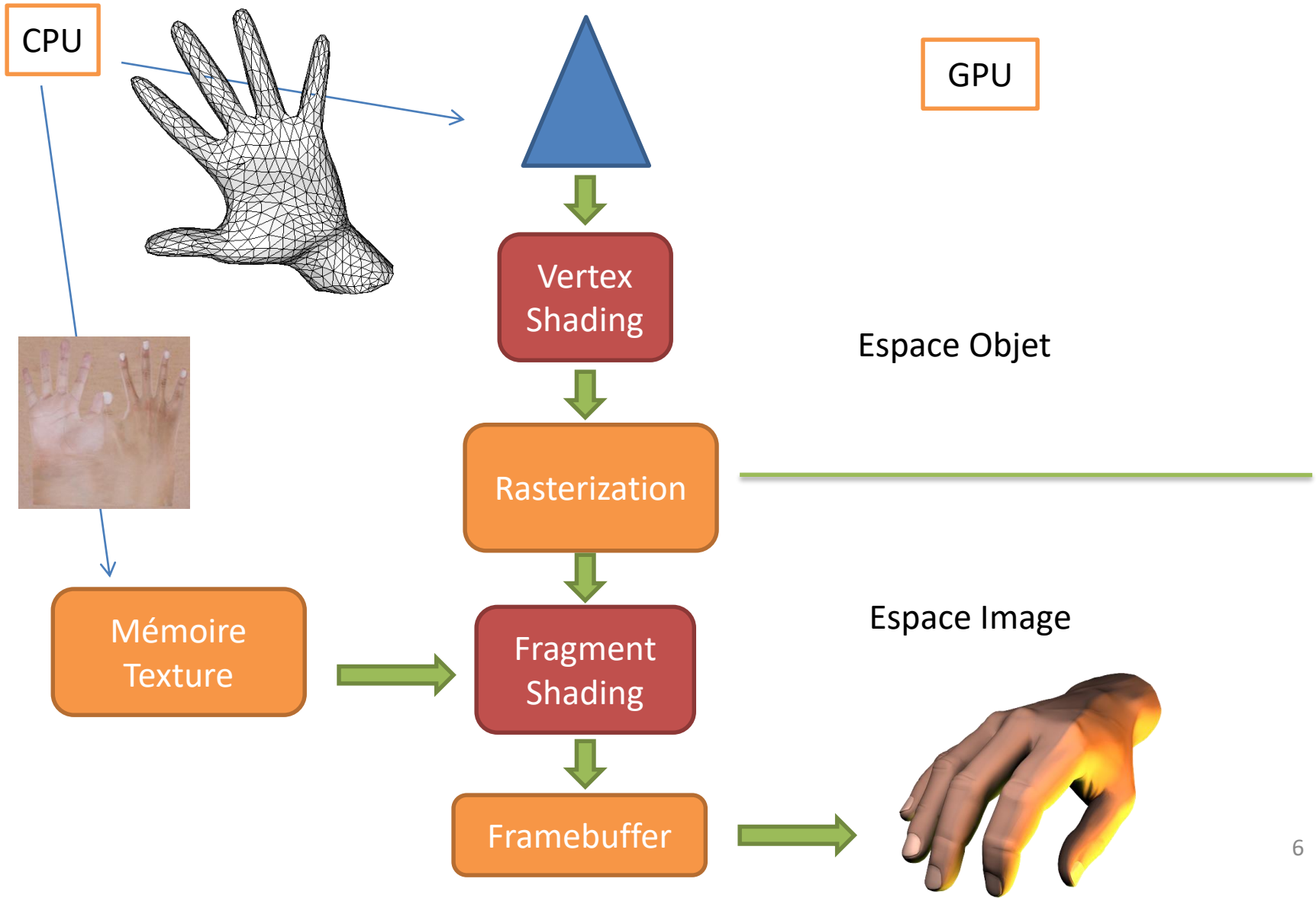
- Graphics Processor Unit
- Conçu pour le calcul 3D en synthèse d'image
- Optimisé pour le rendu par *rasterization*
- Poussé par le marché des jeux vidéo

Rendu



- Rendu = Synthèse d'Image
- Conversion de la scène 3D en une image 2D
- Précalculé ou **temps-réel**
- Spectre: [<empirique>…<physiquement réaliste>]
- Réaliste ou expressif

Rendu GPU par *Rasterization*



Architecture Many-Core

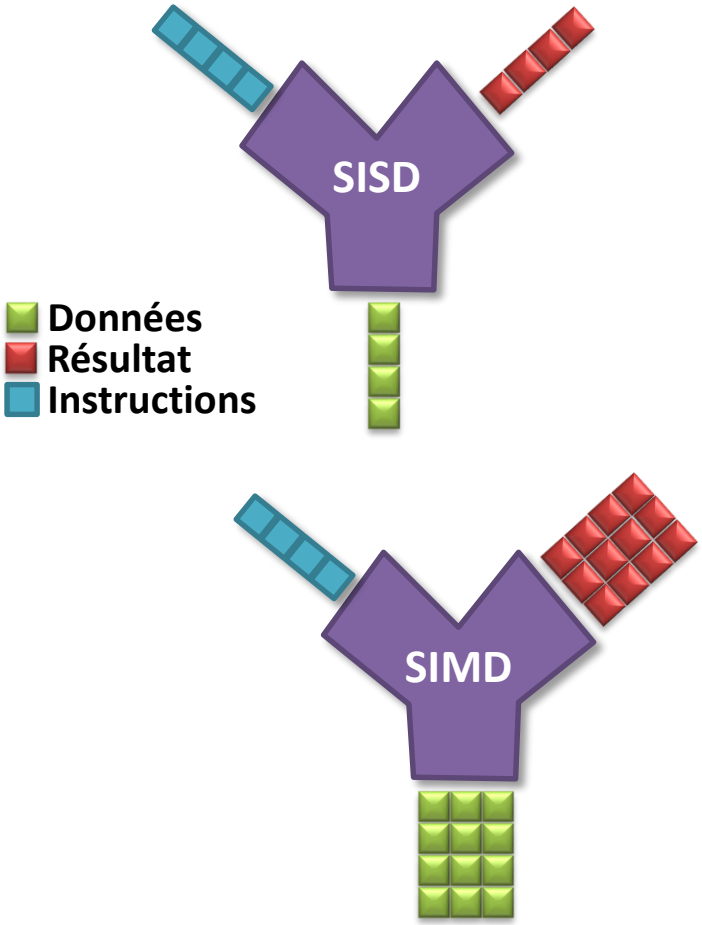
- Point de convergence CPU/GPU

Défis :

- Techniques
  - Coût de production
  - Coût énergétique
- Scientifiques
  - Algorithmique parallèle et calcul distribué
  - Machine hybrides
    - Effets NUMA
    - Flexibilité vs performances
- Pédagogiques
  - Difficile à enseigner en général
  - Cas particulier de la programmation GPU « de base »
    - Langages de Shader
    - GPU Computing (CUDA/OpenCL)

Machine Massivement Multi-Threads

- GPU = centaines de cœurs
- Architecture **SIMD**
  - **Single Instruction / Multiple Data**
- Cœurs limités :
  - Pas d'allocation dynamique de mémoire
  - Pas de pile > pas de récursion
- Hiérarchie de mémoire
  - Effets NUMA
- Bien adapté au calcul intensif « naturellement parallèle »
- Algèbre linéaire efficace



GPU vs Multi-cœur CPU

	CPU (Core i7-965)	GPU (Tesla S1070)
Nombre de coeurs	4	4x240
Puissance brute	70 GFlops	3.73-4.14 TFlops
Bande passante	18	408

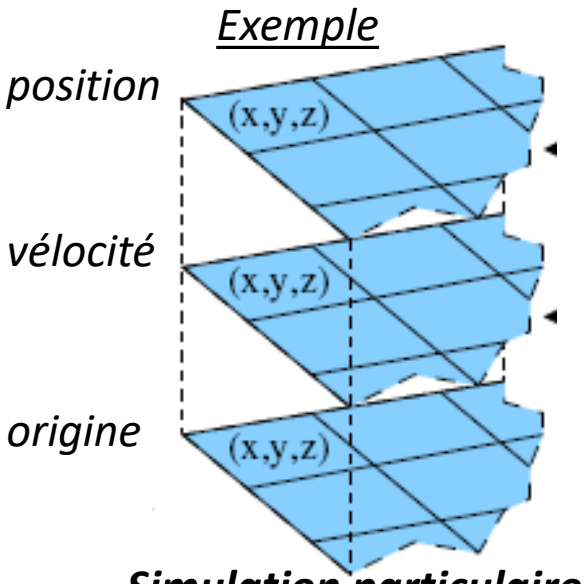
Calcul Général sur GPU

- Utiliser le GPU pour calculer autre chose que des images de synthèse (éq.diff, algèbre linéaire, tris, …)
  - Finance
  - Météo
  - Traitement d'Image/Video/Son
  - Médecine, Chimie, Biologie, Physique, etc

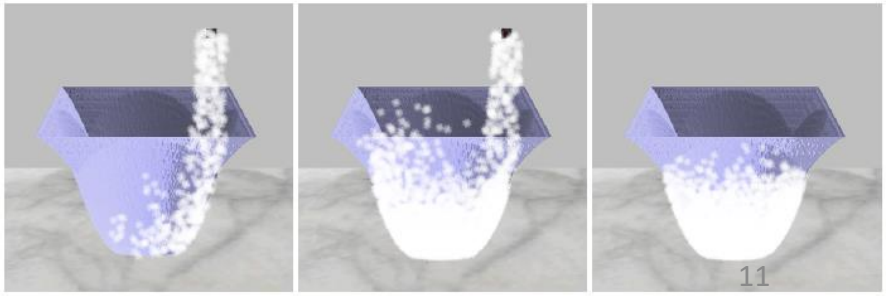
  Accelération observée : 10x à 300x sur les calculs de type
- « GPGPU », General Purpose Computation on GPU, [Mark Harris, 2004]
- 2 approches :
  - ~~Rendu multi-pass hors écran (obsolète)~~
  - Programmation sans contexte graphique (CUDA (Nvidia), Stream Computing (ATI), OpenCL (futur standard))

~~Rendu Multi-Passes Hors Ecran~~

- Un *Map-Reduce* version GPU :
  1. Stocker les données sous forme de textures
  2. Décomposer la fonction à calculer sur les données en fonctions locales, simples, implémentées dans un fragment shader
     $$f(x) := f^0 \circ f^1 \circ \ldots \circ f^n(x)$$
  3. Stocker les paramètres de la fonction dans une texture spéciale P
  4. Générer un quad unique Q, couvrant tout l'écran
  5. Appliquer P à Q, activer f0 et rendre dans un frame buffer spécial F
  6. Mapper F sur P et recommencer l'opération en activant f1
  7. Etc
  8. Rappatrier Fn en mémoire central (puis *Reduce,* éventuellement sur GPU)



Simulation particulaire
Kolb et al. 2006



Programmes sans contexte graphique

- Pure calcul
- GPU comme une grille de processeurs
- Architecture CUDA d'NVIDIA
  - Langage C For CUDA
- Architecture Stream Computing d'ATI
- **OpenCL** : Tentative de normalisation de la programmation

ARCHITECTURE CUDA

Description

- *Compute Unified Device Architecture*
- Architecture matérielle et logicielle des GPU Nvidia
- Programmable via un langage C étendu
- Exploite directement l'architecture unifiée (G80 et +)
  - Pas de contexte graphique
  - 1 cœur graphique = 1 processeur

Principe

- GPU = co-processeur du CPU
- Mémoire propre
- Nombreux threads en parallèle
  - **Kernel** : Un même programme exécuté sur plusieurs threads à la fois
  - Threads très légers (création et activation rapide)
  - Utiliser des **milliers** de threads pour une exploitation optimale

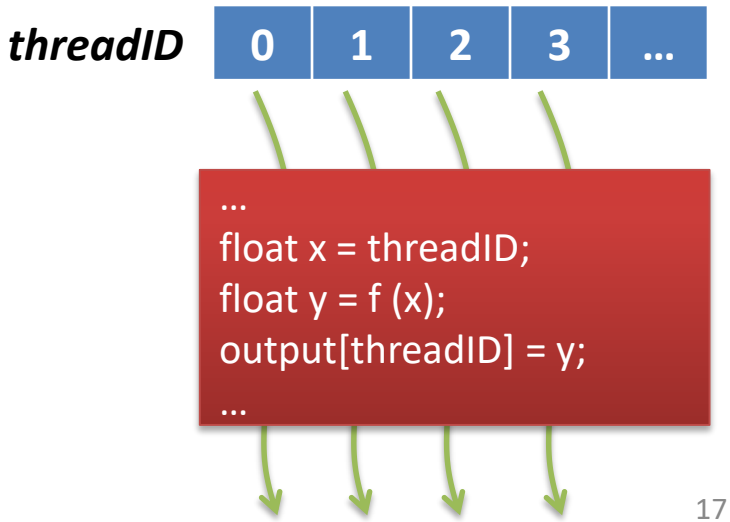
Développement

- Le programmeur développe de petits programmes (noyaux)
  - Centaines de cœurs
  - Milliers de threads
- Pas de gestion explicite de l'ordonnancement des threads, politique mémoire fixe
- Systèmes hétérogènes CPU+GPU
  - Mémoires séparées

Noyau et Threads

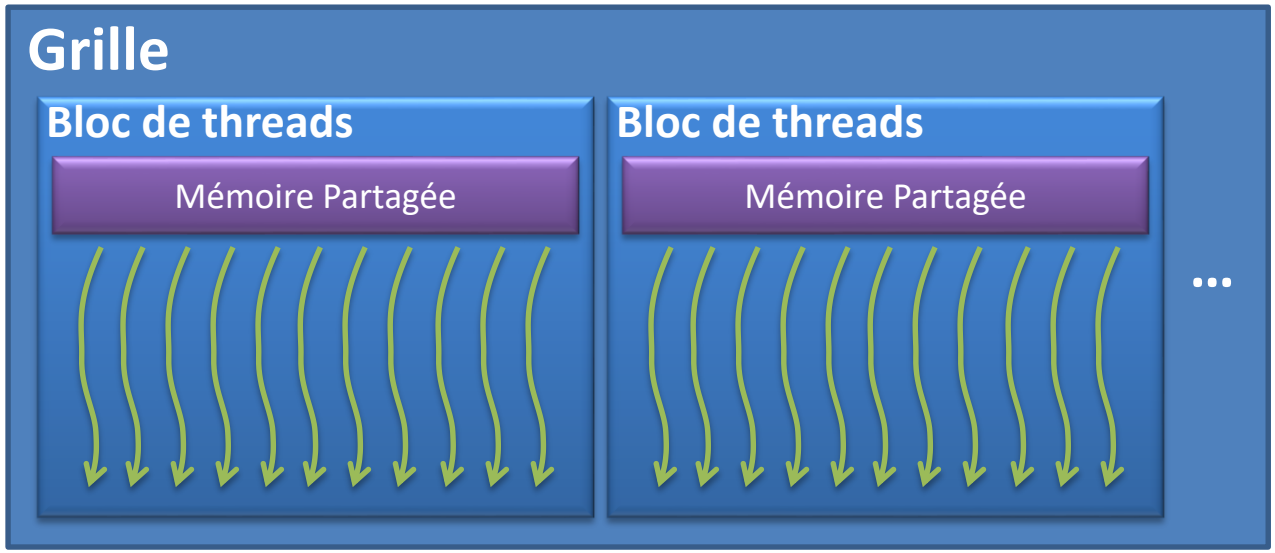
- Noyau : portion de code //
- 1 noyau exécuté à un moment donné sur tous les threads
  - Appelé à évoluer (k noyaux pour n/k threads)
- Particularité de threads GPU vs CPU
  - Très léger à créer
  - Des milliers à la fois (contre quelques uns sur CPU)
  - Executent tous le même code
    - Limitation levée par l'architecture Fermi (2010)

Générique	En Pratique
Hôte (*Host*)	CPU
Périphérique (*Device*)	GPU
Noyau (*Kernel*)	Fonction appelée de l'hôte et exécutée sur le périph.



Coopération entre threads

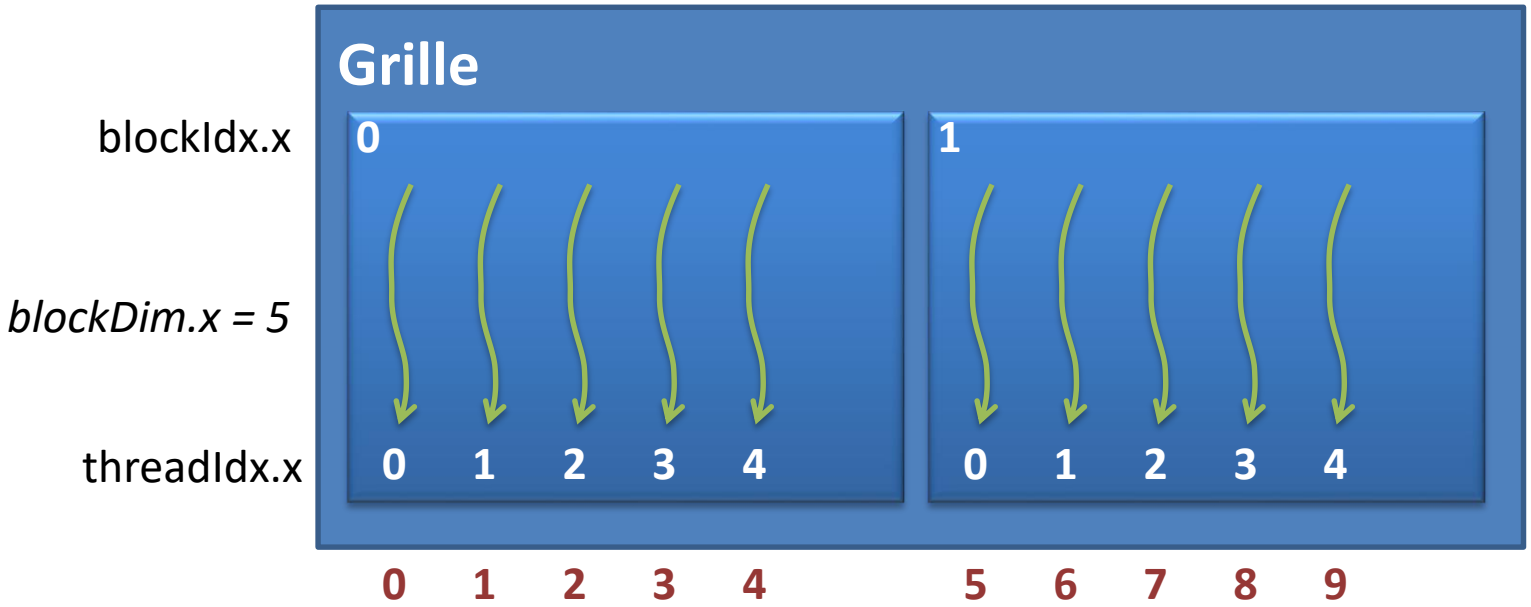
- Partage de résultats intermédiaires
- Factorisation des accès mémoire
- Coopération globale « tout thread vers tout thread » inefficace
- Notion de ***blocs*** de threads
- Un noyau lance une ***grille*** de ***blocs*** de threads



- Communication intra-bloc via la **mémoire partagée**
- Pas de coopération inter-bloc
- Décorrélation code/GPU (#cœurs)

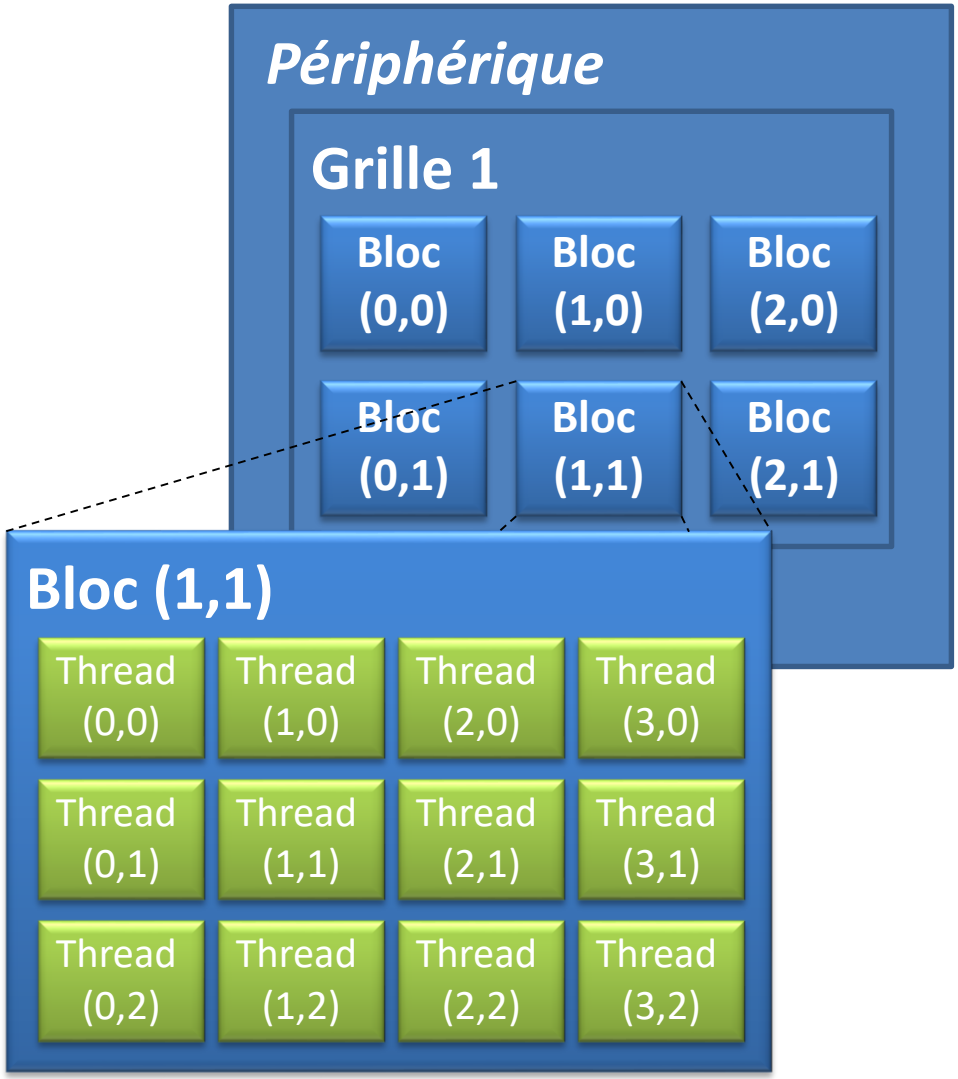
Décomposition de Données

- Chaque thread accède une partie différente d'un tableau
- Informations d'indexation:
  - threadIdx.x : index du thread au sein du bloc
  - blockIdx.x : index du bloc au sein de la grille
  - blockDim.x : nombre de threads par bloc



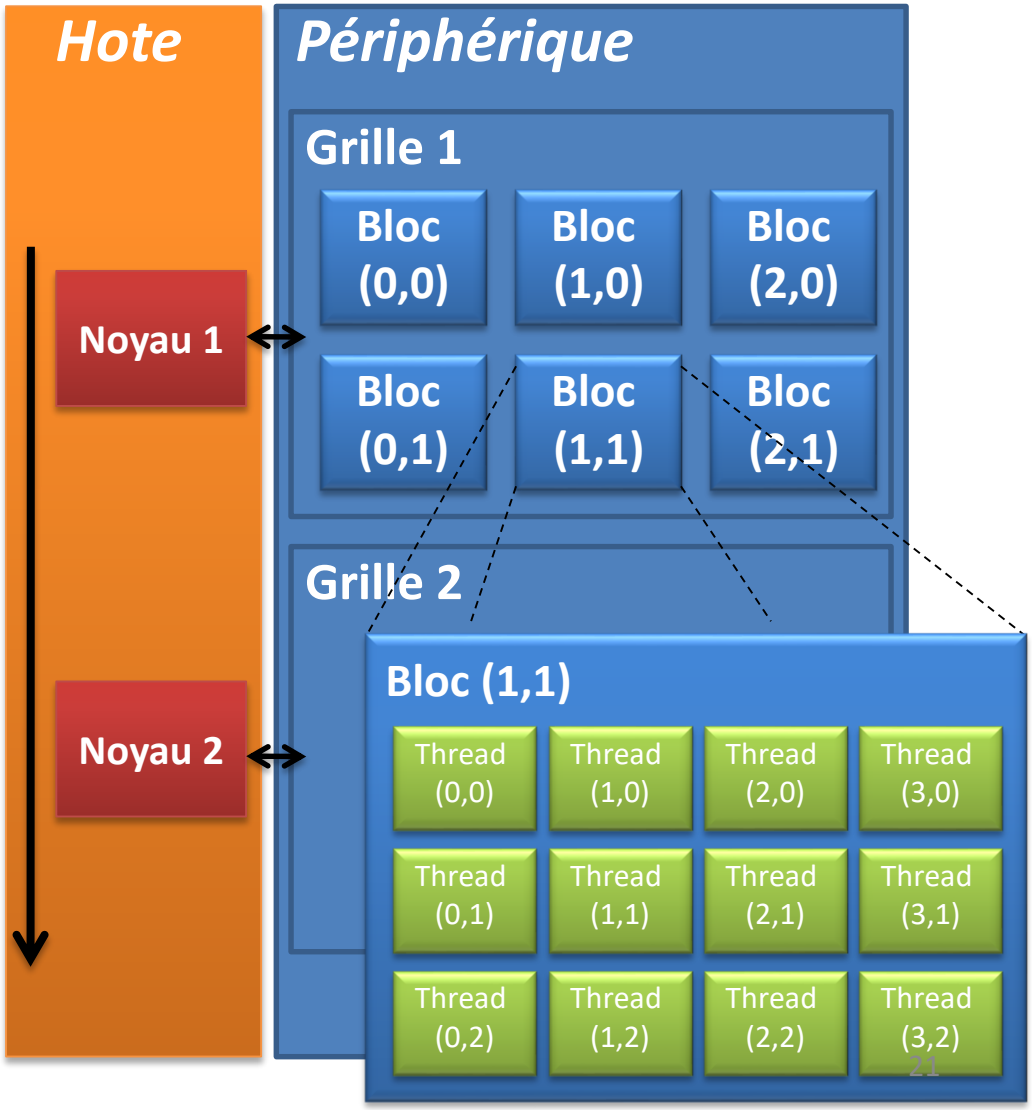
Identifiants multi-dimensionnels

- BlockID : 1D ou 2D
- ThreadID : 1D, 2D ou 3D
- Facilite le traitement de données multidimensionnelles
  - Exemple : traiter une image



Modèle de Programmation CUDA

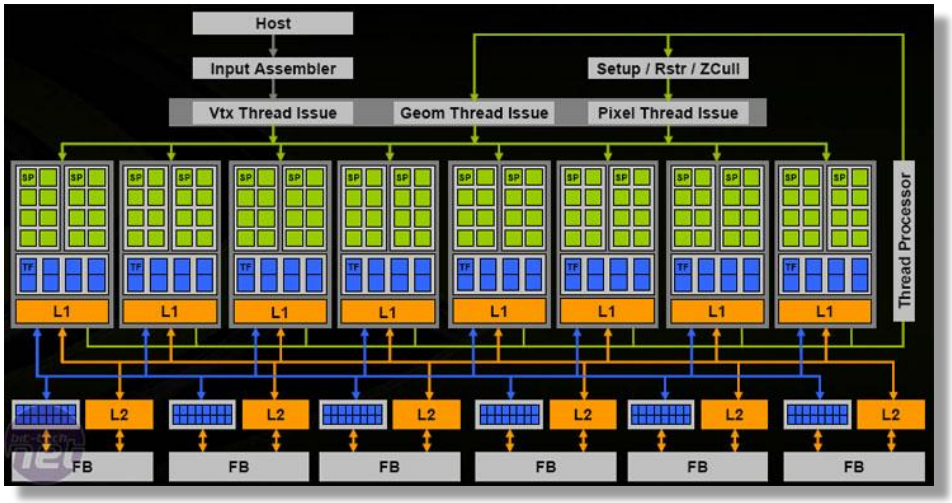
- Noyau exécuté par une grille de blocs de threads
- Bloc de threads:
  - Coopèrent via la mémoire partagée
  - Synchrones
- Threads de différents blocs
  - Pas de coopération



Exemple de Périphérique CUDA

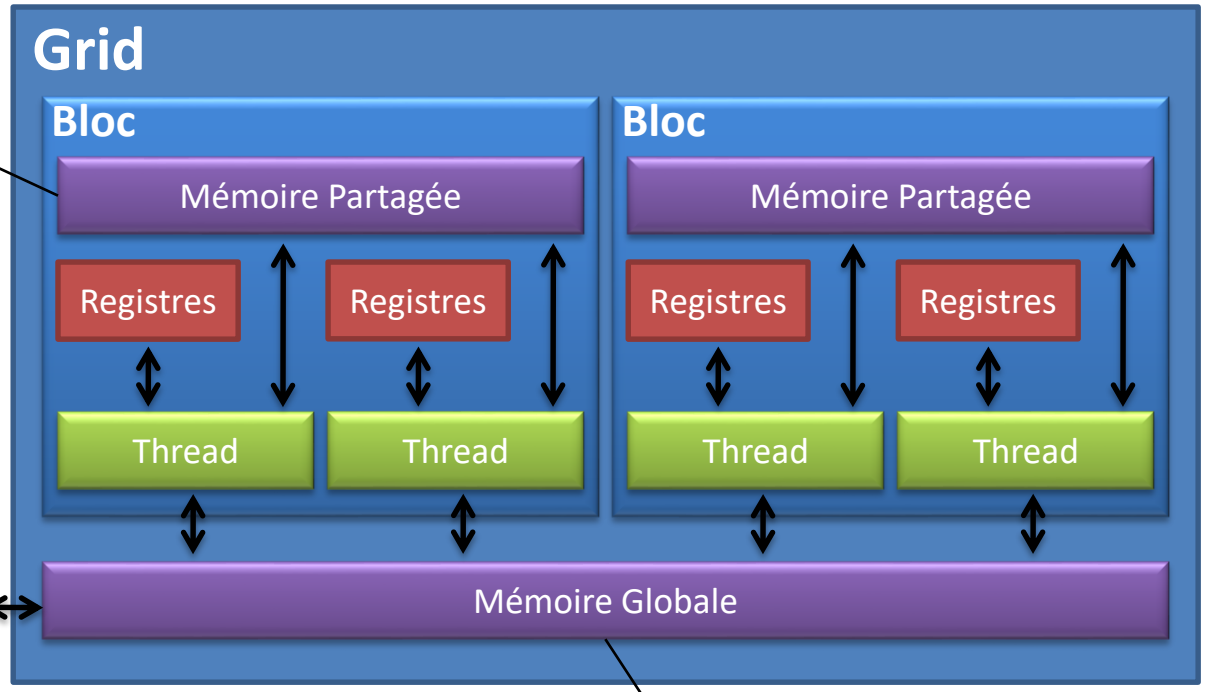
GPU NVIDIA G80

- Les processeurs exécutent les threads
- Le gestionnaire de thread les crée
- 128 processeurs de threads (ou processeur de flux) organisé en 16 multiprocesseurs



Accès mémoire

Partagée par les threads d'un bloc
Sur le processeur
Petite, aussi rapide que les registres



Hote
(Processeur/mémoire centrale)

Lit/écrit uniquement sur la mémoire globale

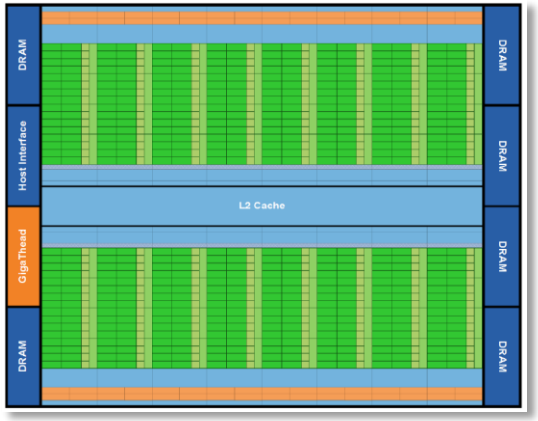
Entrées/sorties des noyau de calcul
Externe au processeur
Massive (2009: 1 à 2 Go)

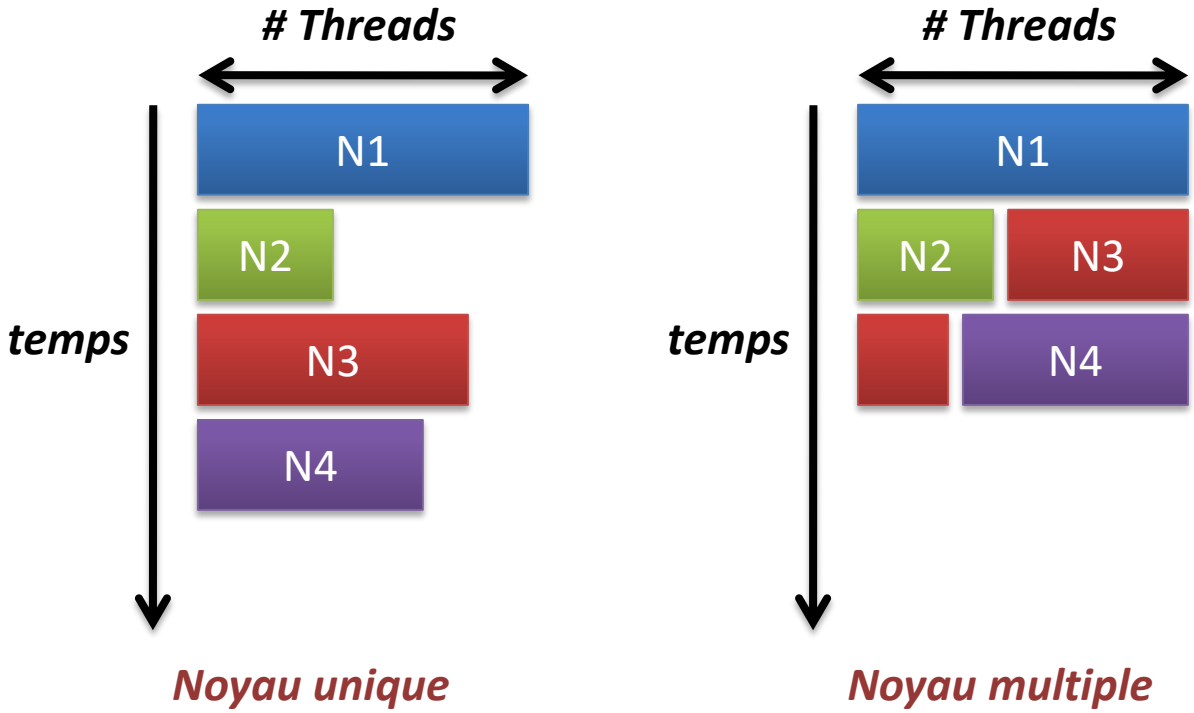
Modèle d'Exécution CUDA

- Noyau lancés sur des grilles
  - GPUs actuel un noyau à la fois
  - Architecture NVIDIA Fermi (2010) : plusieurs noyaux à la fois
- Un bloc exécuté sur un unique multiprocesseur
  - Pas de migration
- Plusieurs blocs concurrents par multiprocesseur
  - Limitations
    - Registres : partitionnés sur tous les threads résidents
    - Mémoire partagée : partitionnée sur tous les blocs résidents

Architecture Fermi

- Hiérarchie de cache
- Plusieurs noyaux à la fois





Architecture Kepler

- Parallélisme dynamique
  - Lancer des noyaux depuis des noyaux

Résumé du Modèle CUDA

- Milliers de threads concurrents très légers
  - Pas de coût d'activation
  - Masque la latence du chargement des instructions et des accès mémoire
- Mémoire partagée
  - Un cache géré par le programmeur-utilisateur
  - Assure la communication et la coopération entre threads
- Accès libre à la mémoire globale
  - Tous les threads peuvent lire et écrire

Mémoire	Situation	Cache	Accès	Permission
Partagée	Sur processeur	N/A	Lecture/écriture	Threads du bloc
Globale	Hors processeur	Non	Lecture/écriture	Tous les threads et l'hôte

PROGRAMMATION CUDA

Bases

- Fonctions spécifiques (en C) pour:
  - Gestion de la mémoire GPU
  - Lancement de noyaux GPU
  - Quelques mots clés et structures spécifiques pour les code GPU
- Eléments supplémentaires:
  - Types *Vector*
  - Synchronisation
  - Vérification des erreurs CUDA
- Dans ce cours : tour d'horizon
- Description exhaustive
  - Programming Guide sur le site NVIDIA

Gestion mémoire

- CPU et GPU ont des espaces mémoire séparés
  - Sémantiquement et physiquement
- L'application CPU gère la mémoire du GPU
  - Allocation/libération
  - Copie vers et depuis le GPU
  - Ne s'applique qu'à la mémoire globale

Allocation/Libération Mémoire

- **cudaMalloc**(void ** pointer, size_t nbytes)
- **cudaMemset**(void * pointer, int value, size_t count)
- **cudaFree**(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Copie de données

- **cudaMemcpy**(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - *direction* définit l cible (CPU ou GPU) de *src* et *dst*
  - Bloque le thread maître CPU: relaché à la fin de la copie
  - Commence à copier lorsque les appels CUDA précédents sont terminés
- enum **cudaMemcpyKind**
  - *cudaMemcpyHostToDevice*
  - *cudaMemcpyDeviceToHost*
  - *cudaMemcpyDeviceToDevice*

Code GPU

- Un noyau (« kernel ») est une fonction C avec quelques restrictions
  - Invoqué par le CPU, 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 statiques
- Les argument des noyau sont copié du CPU au GPU automatiquement

Qualificatifs de fonctions

Qualificatif	Effet
__global__	Invoqué depuis le code CPU, ne peut être appelé depuis le code GPU, retourne « void »
__device__	Appelé depuis les autres fonctions GPU, ne peut être appelé depuis le CPU
__host__	Appelé depuis le CPU et exécuté sur CPU

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

Exécution de noyau

- Syntaxe C d'appel de fonction modifié :

kernel<<<dim3 grid, dim3 block>>>(…)

- Configuration d'exécution ("<<< >>>"):
  - Dimension de la grille: x et y
  - Dimension d'un bloc de threads : x, y, et z

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

Variables pré-définis (GPU)

- __global__ et __device__
- dim3 gridDim;
  - Dimension de la grille (2D au plus)
- dim3 blockDim;
  - Dimensions d'un bloc de thread
- dim3 blockIdx;
  - Index du bloc dans la grille
- dim3 threadIdx;
  - Index du thread dans le bloc

Un noyau minimal

```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}
```

```
__global__ void assign( int* d_a, int value)
{
    Int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Exemple : Incrémenter un tableau

Incrémenter un vecteur à N élément par un scalaire b

Supposons N = 16, blockDim = 4 > 4 blocs

blockIdx.x=0	*blockIdx.x=1*	*blockIdx.x=2*	*blockIdx.x=3*
blockDim.x=4	*blockDim.x=4*	*blockDim.x=4*	*blockDim.x=4*
threadIdx.x=0,1,2,3	*threadIdx.x=0,1,2,3*	*threadIdx.x=0,1,2,3*	*threadIdx.x=0,1,2,3*
idx=0,1,2,3	*idx=4,5,6,7*	*idx=8,9,10,11*	*idx=12,13,14,15*

int idx = blockDim.x * blockId.x + threadIdx.x;

Associera chaque threadIdx à un index global

NB: blockDim toujours>= 32 dans un code réel

Exemple : Incrémenter un tableau

Code CPU

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

Code GPU

```
__global__ void increment_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) );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Exemple SAXPY

```
void saxpy_serial (int n, float a, float * x, float * y) {
        for (int i = 0; i < n; i++)
                y[i] = a*x[i]+y[i];
}
// Appel en série du noyau SAXPY
saxpy_serial (n, 2.0, x, y);
```

Code C

```
__global__ void saxpy_parallel (int n, float a, float * x, float * y) {
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        if (i < n)
                y[i] = a*x[i] + y[i];
}
// Appel parallèle du SAXPY avec 256 thread/block
int nblocks = (n + 255) / 256;
Saxpy_parallel<<<nblocks,256>>> (n, 2.0, x, y);
```

Code CUDA

Lecture/Ecriture

Noyau minimal pour des données 2D

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;
    d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

Synchronisation CPU

- Tous les lancements de noyau sont asynchrones
  - Les appels CPU retournent tout de suite
  - Les noyaux s'exécutent après que tous les précédent appels CUDA 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

Exemple : Code CPU

```
// alloue la mémoire sur le CPU
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);
// alloue la mémoire sur le GPU
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);
// copie les données du CPU au GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
// exécutele noyau
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);
// copie les données du GPU au CPU
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);
// libère la mémoire GPU
cudaFree(d_A);
```

Qualificatifs de variable (Code GPU)

- **__device__**
  - Stockée en mémoire GPU (importante, forte latence, pas de cache)
  - Allocation avec cudaMalloc (__device__ implicite)
  - Accessible par tous les threads
  - Durée de vie : en fonction des besoins
- **__shared__**
  - Stockée sur la mémoire partagée (très faible latence)
  - Allouée en début d'exécution ou à la compilation
  - Accessibles par tous les threads d'un même bloc
  - Durée de vie : exécution du noyau
- Variables non qualifiées:
  - Scalaire et types *vector* sont stockés dans les registres
  - Les tableaux de plus de 4 éléments sont stockés en mémoire GPU globale

Utilisation de la mémoire partagée

Taille connue à la compilation

```
__global__ void kernel(…)
{
    …
    __shared__ float sData[256];
    …
}
int main(void)
{
    …
    kernel<<<nBlocks,blockSize>>>(..);
    …
}
```

Taille connue au lancement du noyau

```
__global__ void kernel(…)
{
    …
    extern __shared__ float sData[];
    …
}
    int main(void)
{
    …
    smBytes = blockSize*sizeof(float);
    kernel<<<nBlocks, blockSize,
    smBytes>>>(…);
    …
}
```

Types *Vector*

Peuvent être utilisés dans les codes CPU et GPU

- [u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4], *double[1..4]*
  - Structures accessible par les champs x, y, z, w :

```
uint4  param;
int y = param.y;
```

- dim3
  - Basé sur uint3
  - Utilisé pour spécifié des dimensions
  - Valeur par défaut (1,1,1)

Synchronisation des threads GPU

- void **__syncthreads();**
- Synchronise tous les threads d'un bloc
  - Génére une *barrière* de synchronisation
  - Barrière standard : tout thread y reste bloqué tant que tous n'y sont pas
  - Utile pour éviter les écrasements (RAW / WAR / WAW) hasardeux lorsqu'on accède à la mémoire partagée
- Autorisé sur code conditionnel seulement s'il est uniforme sur tout le block (ne dépend pas des Idx)

Opérations atomiques sur les entiers

- En mémoire globale GPU:
  - Operations associatives sur entiers signé/non signés
  - **add**, **sub**, **min**, **max**, ...
  - **and**, **or**, **xor**
  - **Increment**, **decrement**
  - **Exchange**, **compare** et **swap**

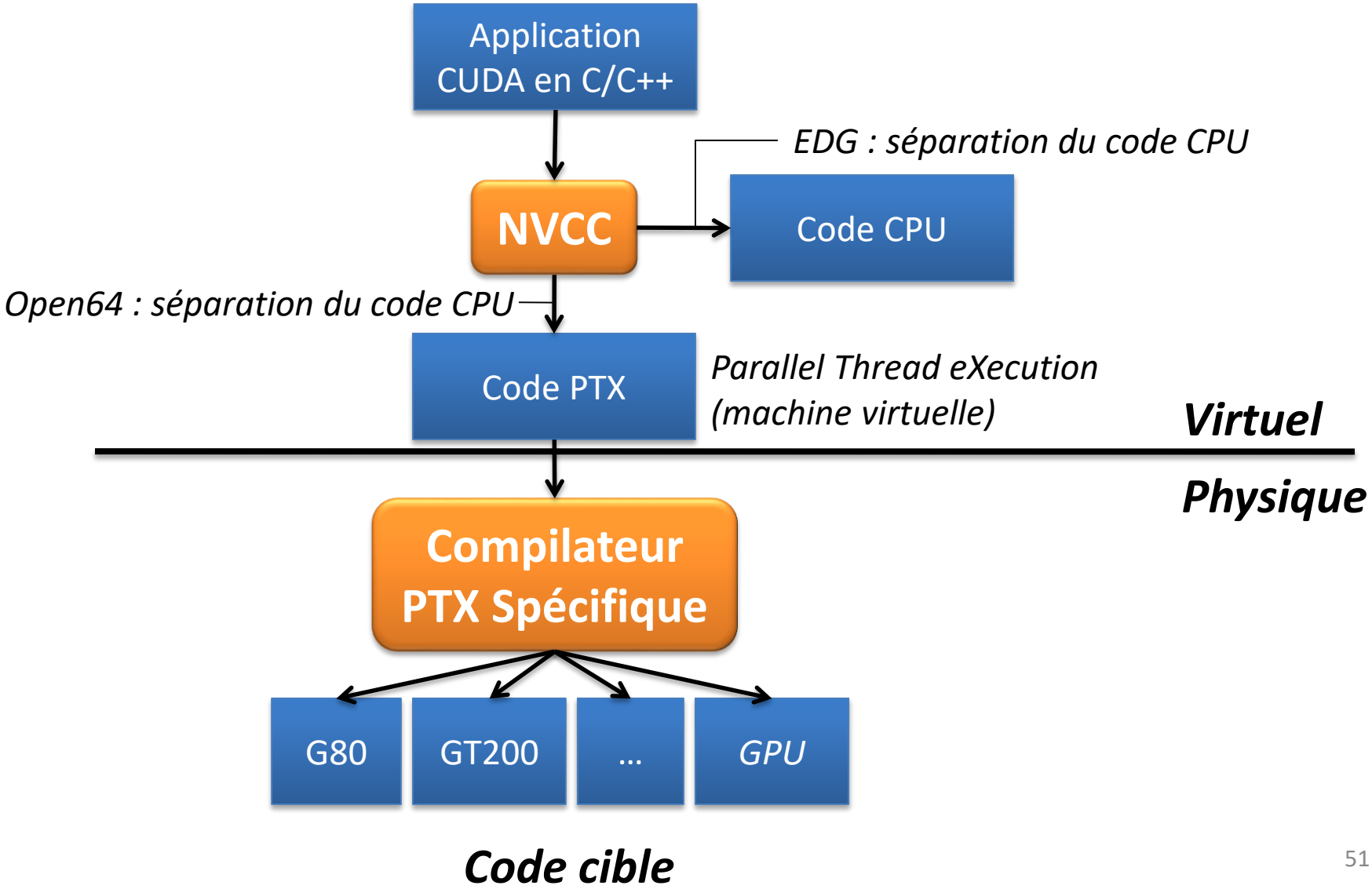
Gestion de erreurs

- Tous les appels CUDA retournent un code d'erreur:
  - Sauf pour les lancements de noyaux
  - **cudaError_t** type
- cudaError_t **cudaGetLastError**(void)
  - Retourne le code la dernière erreur (*no error* a un code)
  - Peut être utilisé pour récupéré l'erreur de l'exécution d'un noyau
- char* **cudaGetErrorString**(cudaError_t code)
  - Retourne une chaine de caractère valide (…\0) décrivant l'erreur

```
Printf ("%s\n", cudaGetErrorString (cudaGetLastError ()));
```

COMPILATION CUDA

Vue d'ensemble



NVCC

- Tout fichier contenant des extension de langage CUDA doit être compilé avec **nvcc**
- NVCC est un meta compilateur
  - Invoque tous les outils nécessaires : **cudacc**, g++, cl, ...
- NVCC peut fournir :
  - Du code C (CPU)
    - Nécéssite une compilation avec un autre outil
  - Directement du code objet PTX
- Un exécutable contenant du code CUDA nécessite:
  - CUDA core library (**cuda**)
  - CUDA runtime library (**cudart**)
    - Si l'API runtime est utilisée
    - Charge la bibliothèque CUDA

Bibliothèques CUDA

- Cublas
- Cuda FFT
- CuSPARSE
- CuRAND
- etc

OPENCL

OpenCL I

- GPGPU Standardisé
  - Khronos Group (OpenGL, SVG, OpenAL, etc)
  - **Très similaire à CUDA**
    - La plupart du temps, un simple renommage
      - Bloc >> workgroup
  - Supporté par tous les constructeurs
  - Calcul hétérogêne : GPU et/où multicore CPU
    - *StarPU*
  - Portabilité depuis CUDA simple
  - Utilisable aussi pour le calcul multi-cœur/CPU
    - INTEL OpenCL SDK

OpenCL II

- OpenCL C : C99 avec extension
- Ensemble de fonctions dédiées au calcul fournit en standard
- Standard : définition matériel et précision numérique
- Définit la notion de programme en plus de la notion de noyau
  - Ensemble de noyau + code non //
  - Une bibliothèque dynamique
  - File d'exécution

SAXPY OpenCL

```
__kernel void saxpy_gpu (const int n,
                         const float a,
                         __global const float* x,
                         __global float* y) {
    const int idx = get_global_id (0);
    if (idx < n)
        y[idx] = a*x[idx] + y[idx];
}
```

ETUDE DE CAS

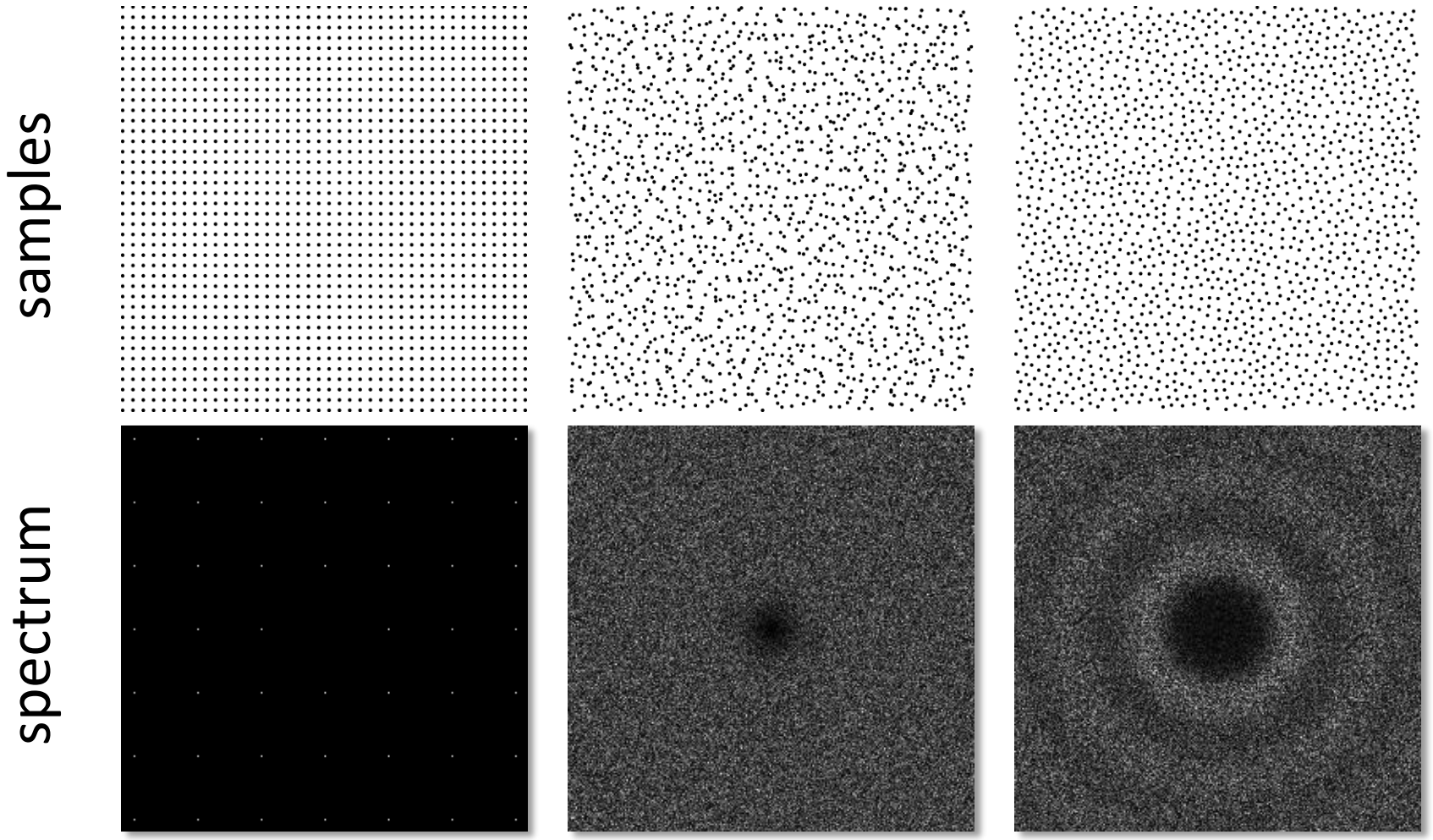
Cas

- Cas de la programmation GPU graphique:
  - GLSL et rendu temps-réel
    - Eclairage Global
    - Tessellation
- Cas de la programmation CUDA
    - Echantillonnage en disques de Poisson
    - Problème à N corps

Parallel Prefix Sum (SCAN)

Poisson Disc Sampling

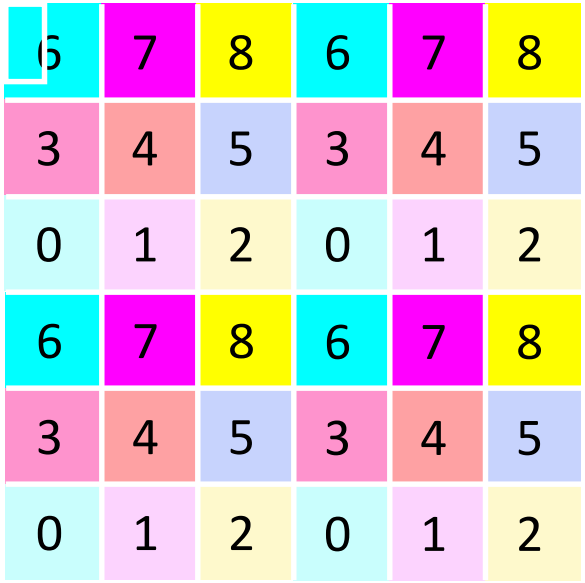
Wei, SIGGRAPH 2008



Poisson Disc Sampling

Wei, SIGGRAPH 2008

grid partition
scanline order



O easy to compute
X bias! (scanline)

random partition

O good quality
X hard to compute
(sequential)

grid partition
random order

O easy to compute
O good quality

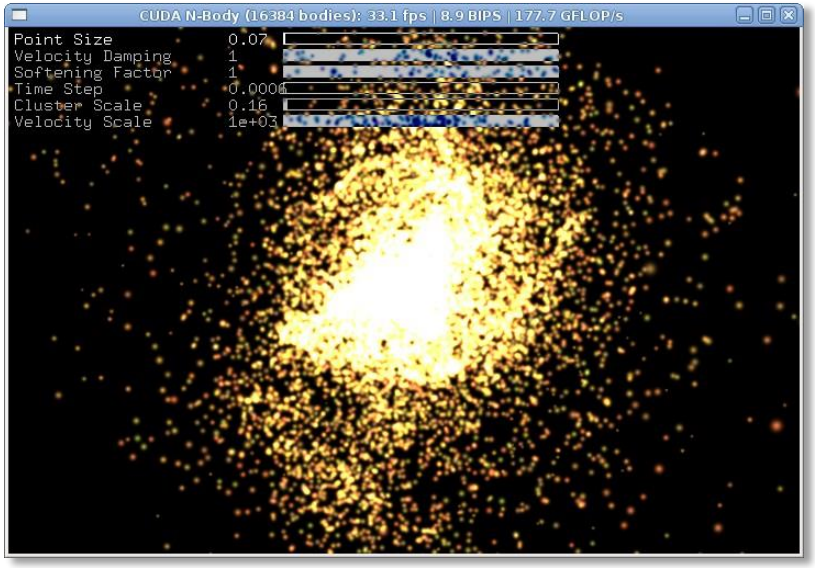
Poisson Disc Sampling

Wei, SIGGRAPH 2008

O: à la volée ***P***: pré-calculé

	# Echantillons par seconde				
	2D	3D	4D	5D	6D
O **GPU** (NVIDIA 8800 GTX)	4.06 M	555 K	42.9 K	2.43 K	179
O Boundary sampling [Dunbar & Humphreys 2006]	0.20 M	X	X	X	X
O Hierarchical dart throwing [White et al. 2007]	0.21 M	X	X	X	X
P Wang tiling [Kopf et al. 2006]	1 ~ 3 M	X	X	X	X
P Polyominoes [Ostromoukhov 2007]	>1 M	X	X	X	X

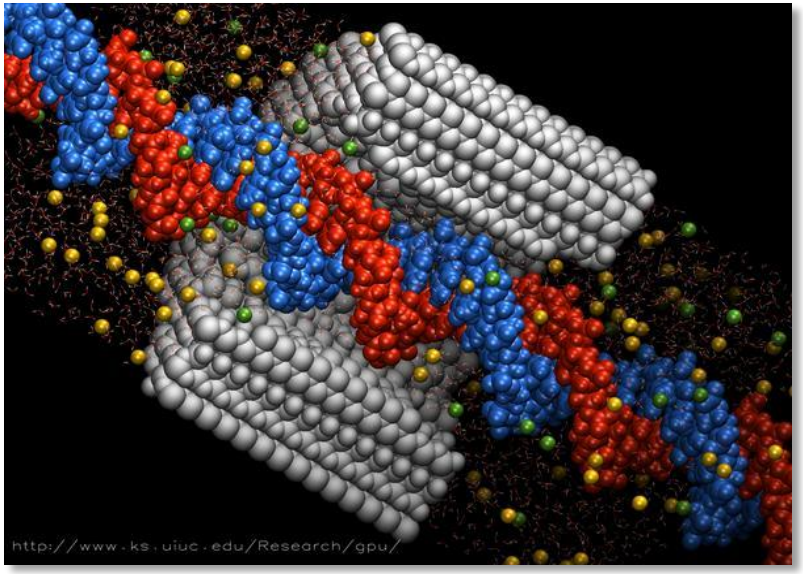
Applications Scientifiques



Fast N-Body Simulation,

Harris,

GPU Gems 3



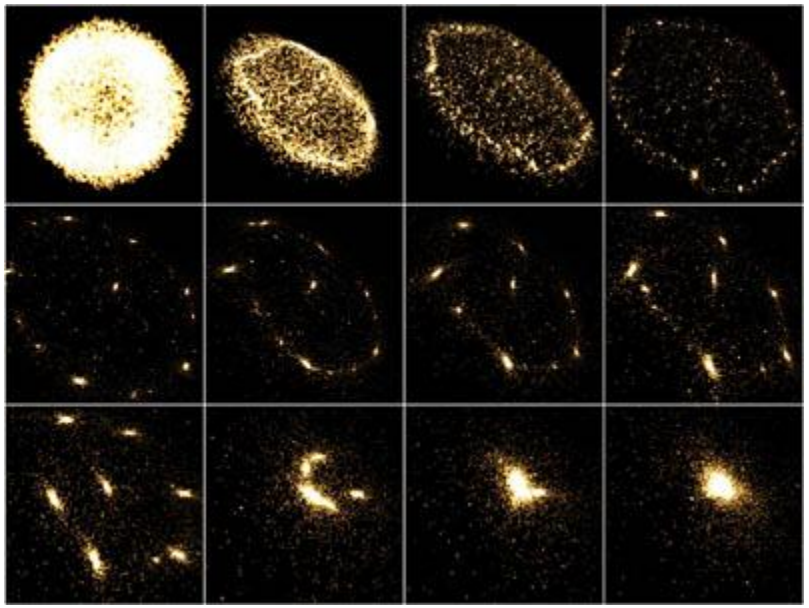
DNA though Synthetic Nanophore,

Theoretical and Computational Biophysics Group,

UIUC

Problème à N Corps I

- Approximation numérique d'un système de N corps interagissant en permanence et de manière continue
- Problème fréquent
  - simulation astrophysique
    - Un corps = une galaxie oue une étoile
    - Chaque corps exerce une force gravitationnelle sur tous les autres
  - Repliement de protéines
    - Forces exercées : force électrostatique et force de Van der Waals



Problème à N Corps II

- Solution *brute force* :
  - Considérer toutes les pair >> complexité O(N^2)
- Solution approchée :
  - ne considérer que les pairs que pour les corps « suffisamment » voisins
  - Approximer les forces issues de corps distants par de grands champs
  - Le calcul sur les pairs proche reste couteux
    - Accélération GPU

Problème à N Corps III

A chaque pas de temps t, la force subit en un corps i (définissant sont accélération en fonction de sa masse) est

$$F_i = \sum_{j \neq i} f_{ij}$$

Donnée : description d'un corps
= {acc , masse} ou {force ,masse}.

Avec f les force de paires
Parallélisation triviale : matrice NxN stockant les interaction par pair >> O(N^2) mem
Approche par ***sérialisation***
Découpage en *tiles* de taille pxp (p^2 interactions) : 2p description de corps nécessaires, p réutilisable

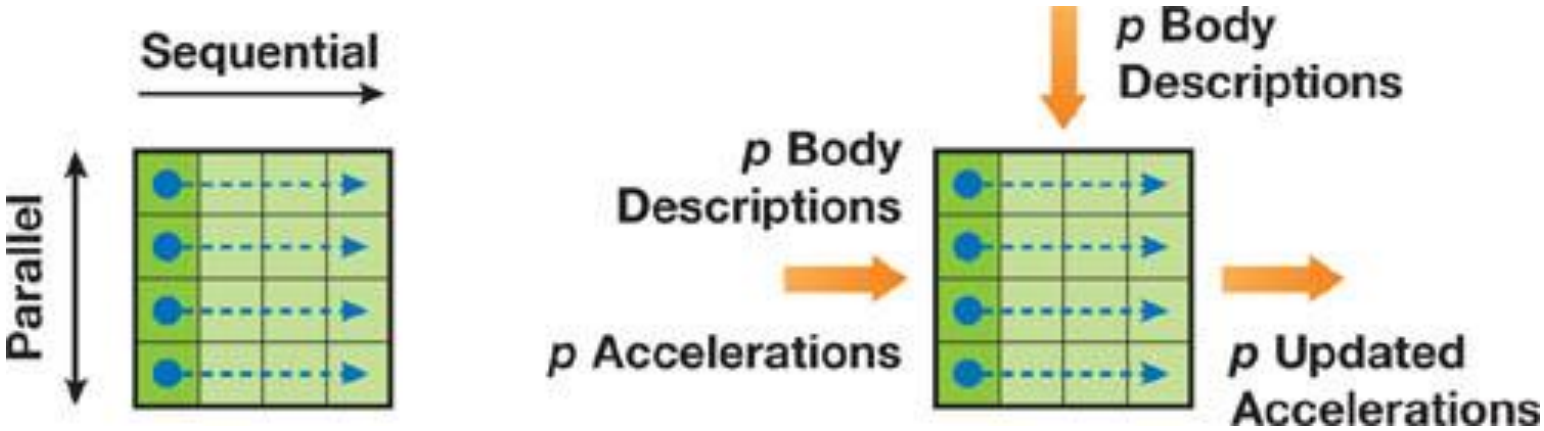
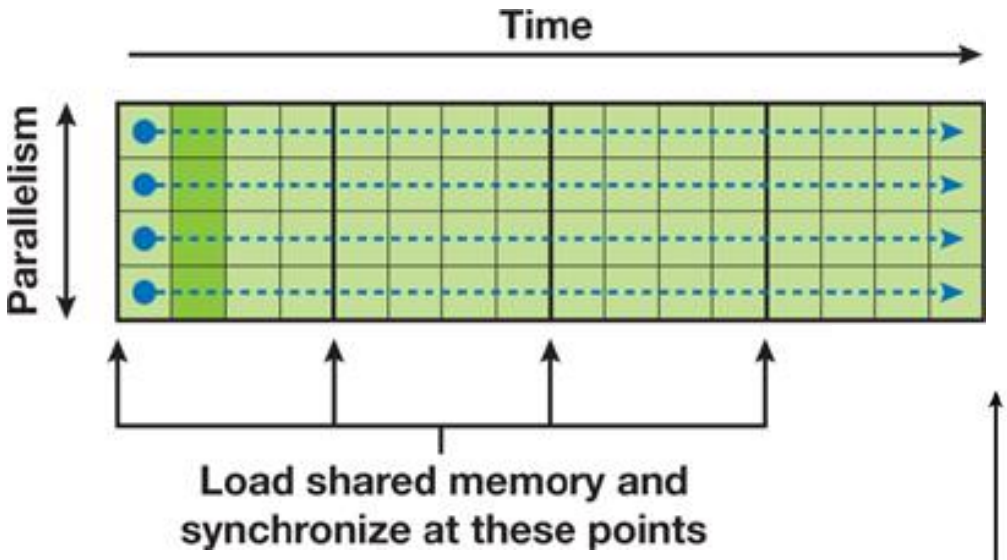
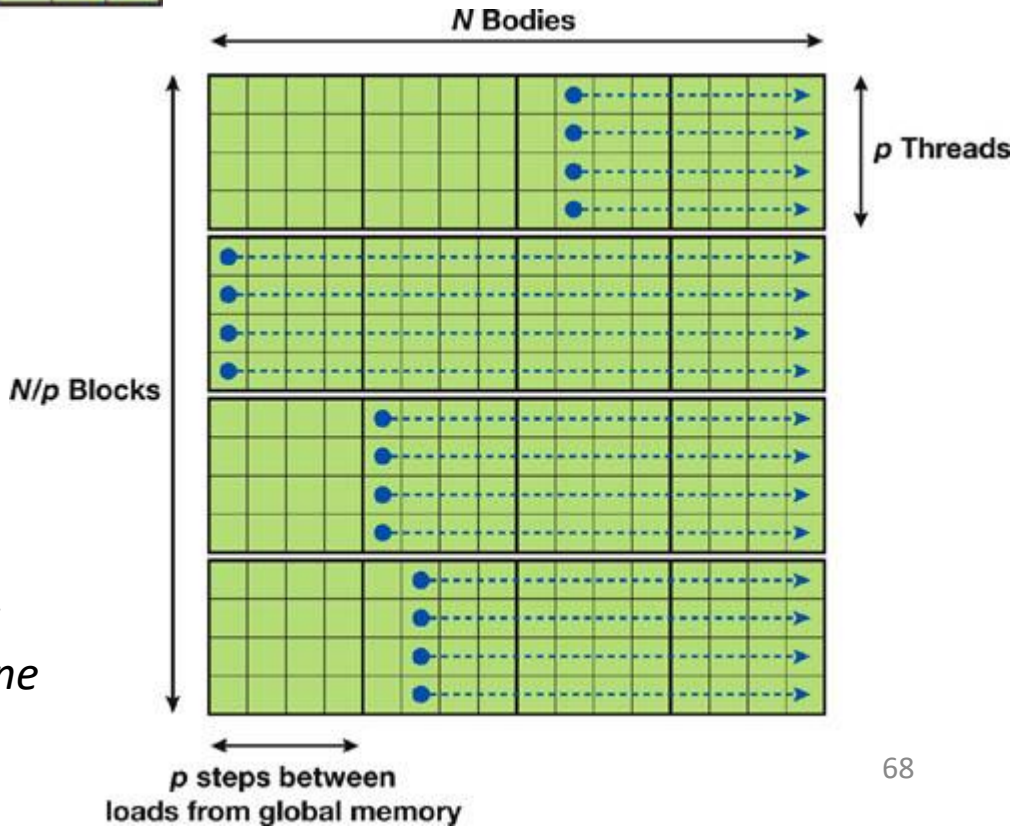


Image Nvidia/Addison Wesley – GPU Gems 3

Problème à N Corps IV





Images Nvidia/Addison Wesley – GPU Gems 3

Nombreuses optimisations possibles. Voir *Fast N-Body Simulation with CUDAn Nylan et al., GPU Gems 3 (version html gratuire en ligne sur developer.nvidia.com)*

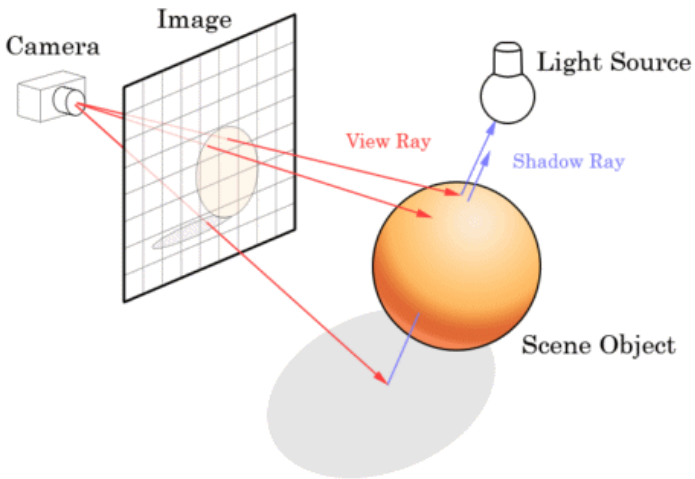
DISCUSSION

Evolution de l’architecture et unités dédiées

- Tensor cores (deep learning)
- RT Cores (physically-based rendering)

Evolution de la Programmation Graphique des GPU

- Compute Shaders
  - Évite l'interaction complexe OpenGL-CUDA/OpenCL
- Lancer de rayon
  - Optix/RTX/DXR
- API modernes: Vulkan, Metal, DX12



Références

- developer.nvidia.com
- developer.amd.com
- OpenCL Spécifications, Khronos