Architecture des GPU

(GPU=Graphics Processing Unit)

gael.guennebaud@inria.fr

http://www.labri.fr/perso/pbenard/teaching/pghp

Plan du cours

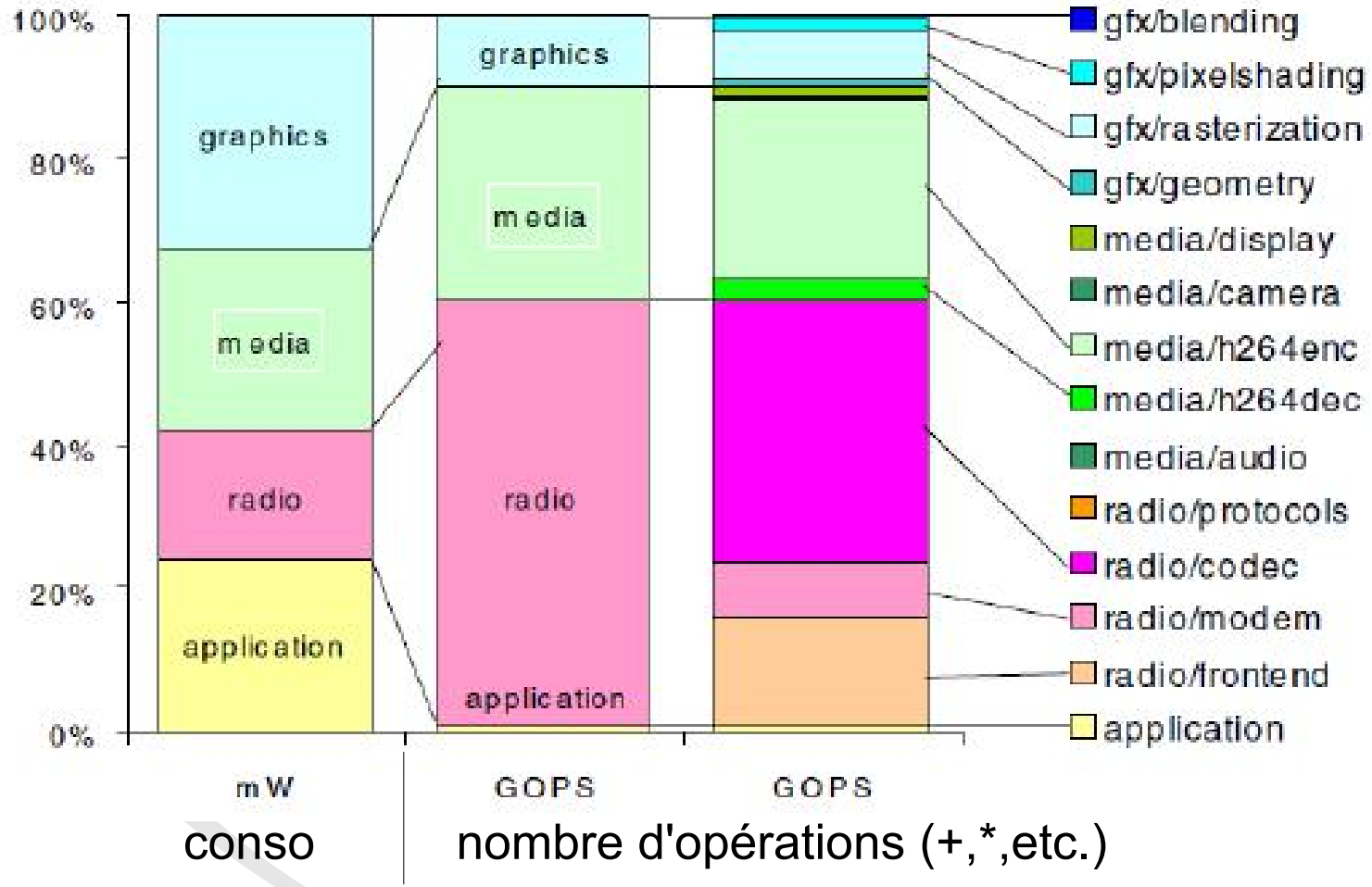
- **Motivations pour les GPUs**
  - single core → multi-core → many-core
- **Architecture des GPUs**
  - CPU versus GPU
- **Programmation des GPUs pour des applications non graphique**
  - briques de bases pour la programmation many-core
  - CUDA

Motivations pour le GPU

- **Toujours plus de puissance...**
  - plus rapide : de x10 à x100 ou plus !!
  - résoudre des problèmes plus gros
  - plus de précisions
  - rendre possible de nouvelles applications, nouveaux algorithmes
  - réduire la consommation
  - etc.

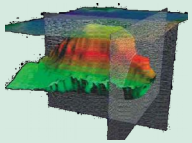
Code optimisé et consommation énergétique

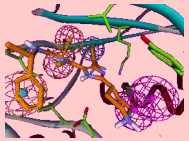
- **Exemple sur un smartphone :**

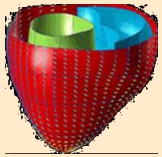


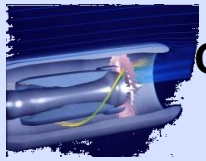
Domaines d'applications des GPU

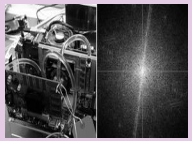
- **Initialement**
  - Synthèse d'Images
- **Applications généralistes, exemples :**
  - Traitement d'images
  - FFT
  - Simulation
    - par éléments finis
    - système de particule
  - Algèbre linéaire, optimisation
  - Algorithmes de trie, de recherche ....
  - etc.

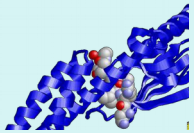

Computational Geoscience

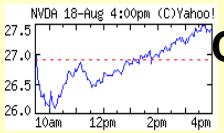

Computational Chemistry


Computational Medicine


Computational Modeling


Computational Physics


Computational Biology



Computational Finance

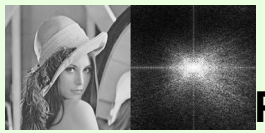


Image Processing

!! vidéos !!

Comment augmenter les performances ?

→ Améliorer les algorithmes

(voir autres cours)

→ **Augmenter la puissance de calcul ?**

→ **Comment ?**

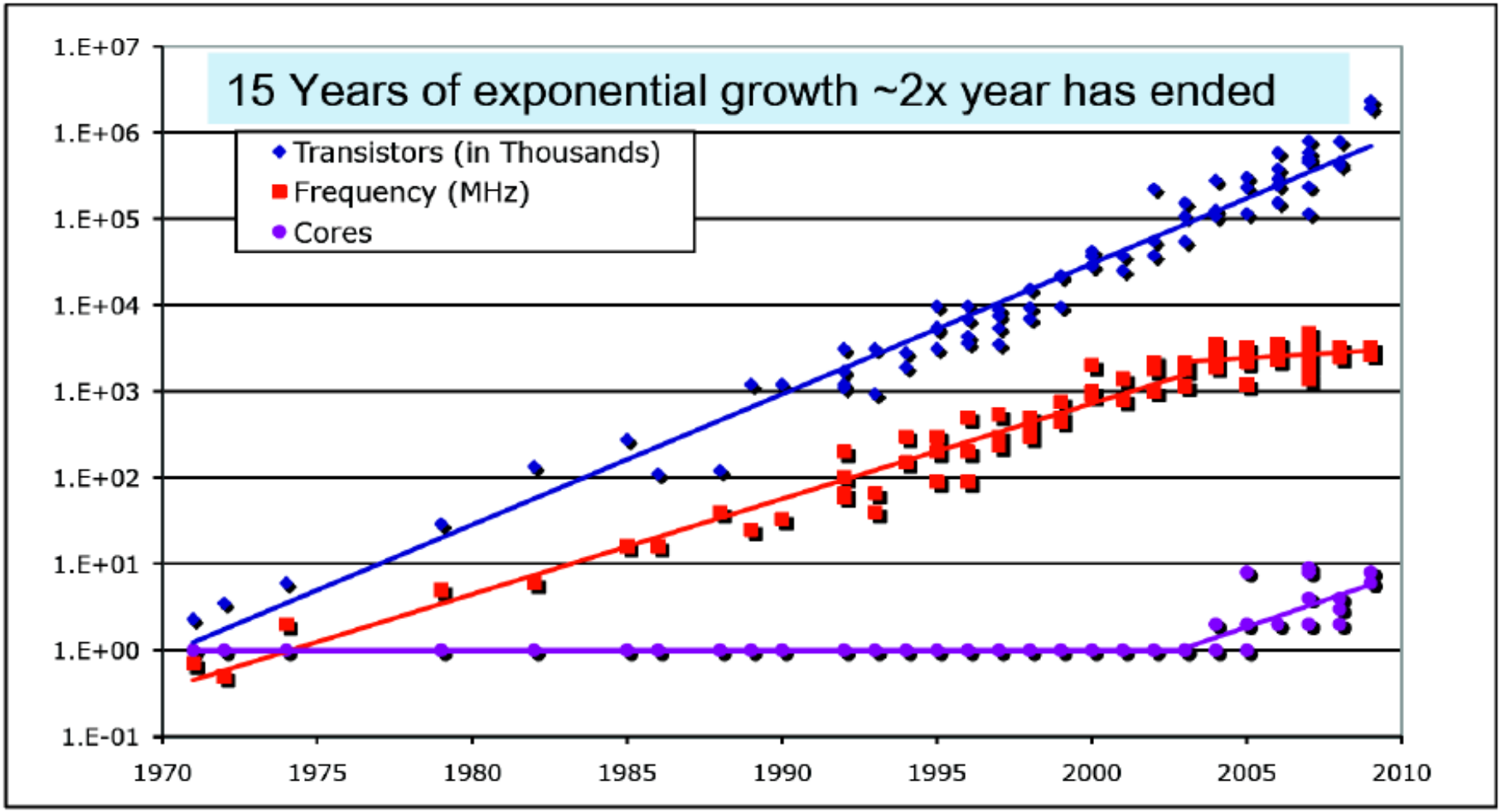
→ **Comment exploiter cette puissance ?**

Exemple de puissance brute

- top 5 des supercalculateurs : de 10000 à 55000 Tflops
- 1 GPU : ~5 TFlops == top 5 en 2001 !

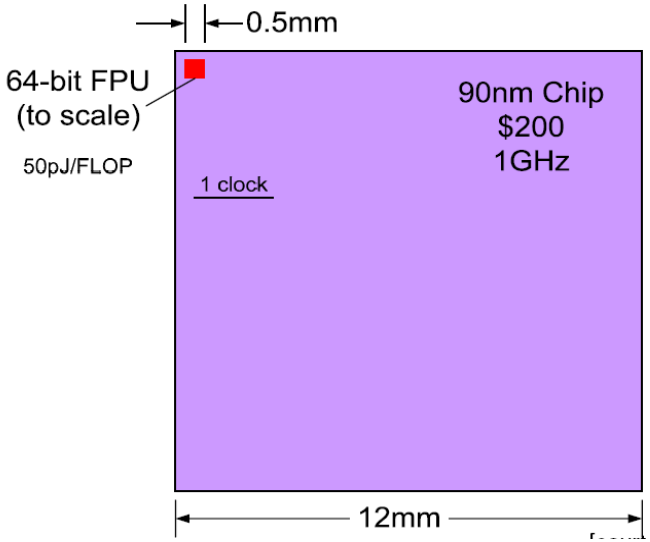
Loi de Moore...

- **Le nombre de transistors qui peut être intégré facilement dans un microprocesseur double tout les deux ans**



Motivations pour le multi-cores

- **Finesse de gravure**
  - 32nm en 2010, 22nm en 2012, 14nm en 2014, …
  - demain : 10nm
    → plus de transistors par circuit



- **Leviers pour augmenter les performances**
  - avec un seul processeur :
    - augmenter la fréquence ?
      - difficile, consommation++, chaleur++
    - augmenter la complexité des processeurs
      - opérations spécialisées, logique (ex. prédiction de branchement), cache, etc.
      - x2 logic → x1.4 performance (Pollack's rule)

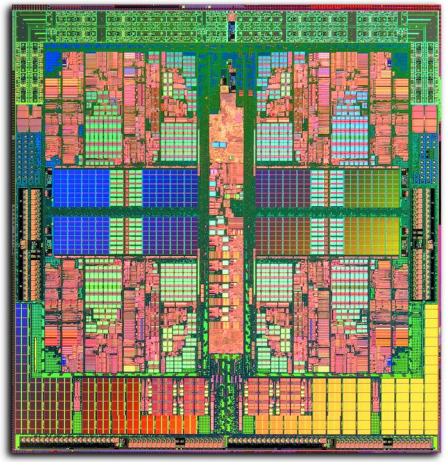
!! vidéo !!

Motivations pour le multi-cores

- **Leviers pour augmenter les performances (cont.)**
  - multiplier le nombre d'unités de calcul :
    - parallélisme au niveau des instructions
      - *out-of-order execution, pipelining*
        → difficile (beaucoup de logique) et limité
    - parallélisme au niveau des données (SIMD)
      - une seule instruction appliquée sur plusieurs registres
      - unités vectorielles : SSE, AVX, NEON, Cell SPE, (GPU)
      - efficace pour certaines applications, mais relativement difficile à exploiter et reste limité

un seul coeur

    - **parallélisme au niveau des <u>threads</u>**
      - mettre plusieurs processeurs cote-à-cote sur un même chip
        - multi-core, many-core (>100)
      - multi-processor : plusieurs chip sur une même carte mère



Motivations pour le multi-cores

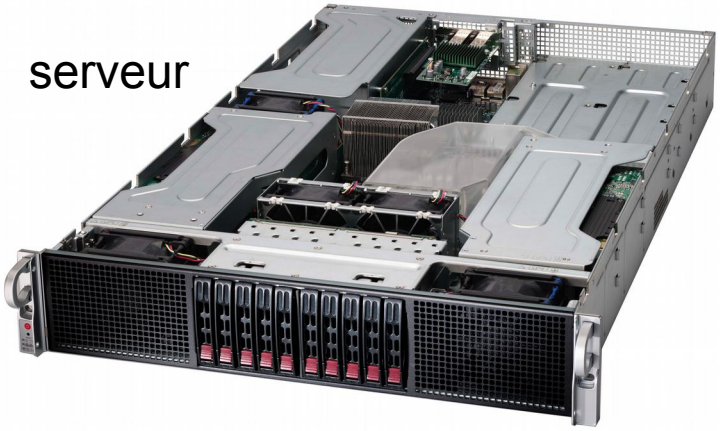
- **Plusieurs processeurs par circuits, plusieurs circuits par carte**
  - réels gains théoriques:
    - 2 « petits » processeurs → x1.8 perf
    - 1 processeur 2 fois plus gros → x1.4 perf
  - consommation et dissipation de chaleur réduite
    - *N* processeurs légèrement plus lents consomment autant qu'un seul rapide
    - activation individuelle des processeurs
  - accès mémoires
    - plusieurs CPU peuvent accéder en même temps à la mémoire
    - permet d'augmenter la bande passante
      - même sans en réduire la latence
  - simplicité
    - plus simple de concevoir et fabriquer pleins de « petits » CPU simples, qu'un seul gros et complexe
      → améliore également la robustesse et absence de pannes/bugs

Motivations pour le multi-cores

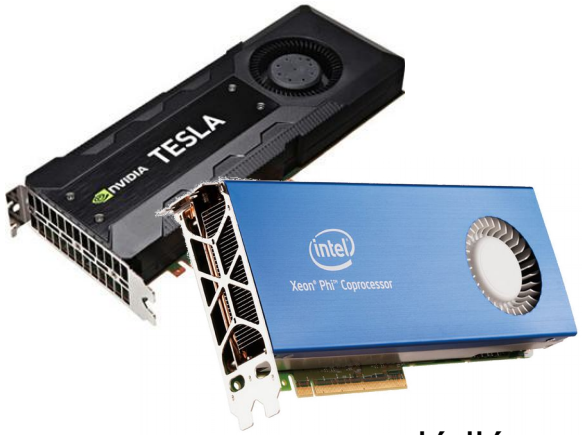
→ ***multi-core everywhere***

- *CPU*
- *GPU*
- *super-calculateur*
- *systèmes embarqués*
- *smart-phones*

serveur



carte graphique
(GPU :1000-3000 coeurs)



co-processeur dédié
(GPU ou centaine de CPUs)

embarqué (ex. Jetson)
(4 cœurs CPU
+ 200-300 cœurs GPU)
[↔ supercalculateur en 2000]

mais…

- **Programmer est difficile**
- **Programmer parallèle est encore plus difficile**
  - Trouver le parallélisme : trouver des tâches pouvant être exécutées en même temps (→ démo/exemple)
  - Coordination entre les tâches (avoid races and deadlocks)
    - éviter les attentes

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

race condition
(résultat différent en fonction de l'ordre d'exécution)

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

correct behavior
(critical section)

mais…

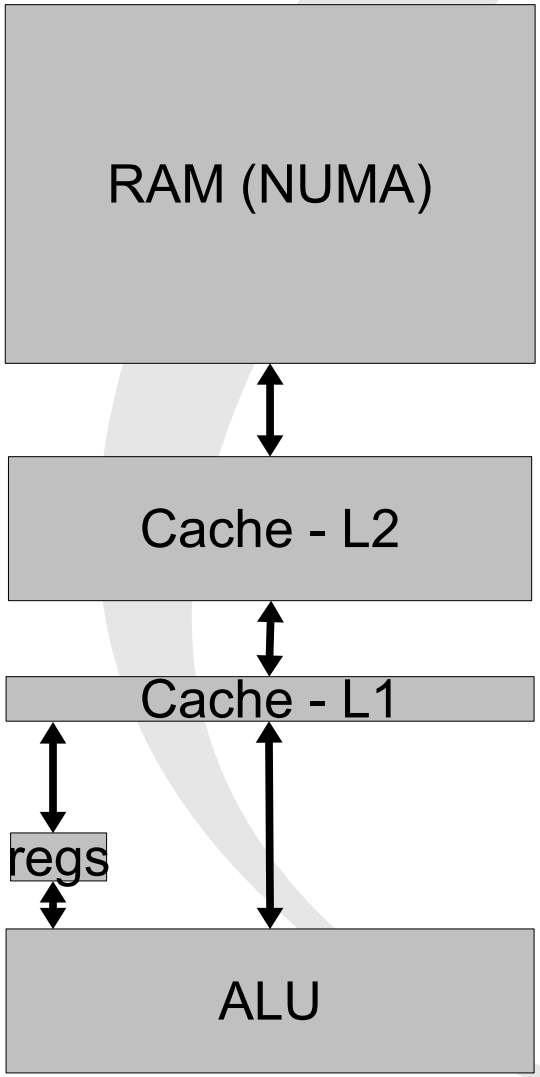
- **Programmer est difficile**
- **Programmer parallèle est encore plus difficile**
  - Trouver le parallélisme : trouver des tâches pouvant être exécutées en même temps (→ démo/exemple)
  - Coordination entre les tâches (avoid races and deadlocks)
    - éviter les attentes
  - Contrôler les surcouts
    - limiter les échanges de données
    - limiter les calculs supplémentaires
- **Modèles simplifiées**
  - Google's « map-reduce » → divide and conquer
  - **Nvidia's data parallelism → big homogeneous array**
  - Erlang's fonctional programming → side-effect free
  - ...

Rappels

-

Architecture des CPUs

Rappel CPU - Hiérarchie mémoire



x1000 bigger ; ~400 cycles

x100 bigger (900x900 floats) ; 40-100 cycles

x100 bigger (90x90 floats) ; 1-4 cycles

small (8x8 floats) ; 1 cycle

CPU - Parallélisme

- **3 niveaux de parallélisme :**

  1 – parallélisme au niveau des instructions

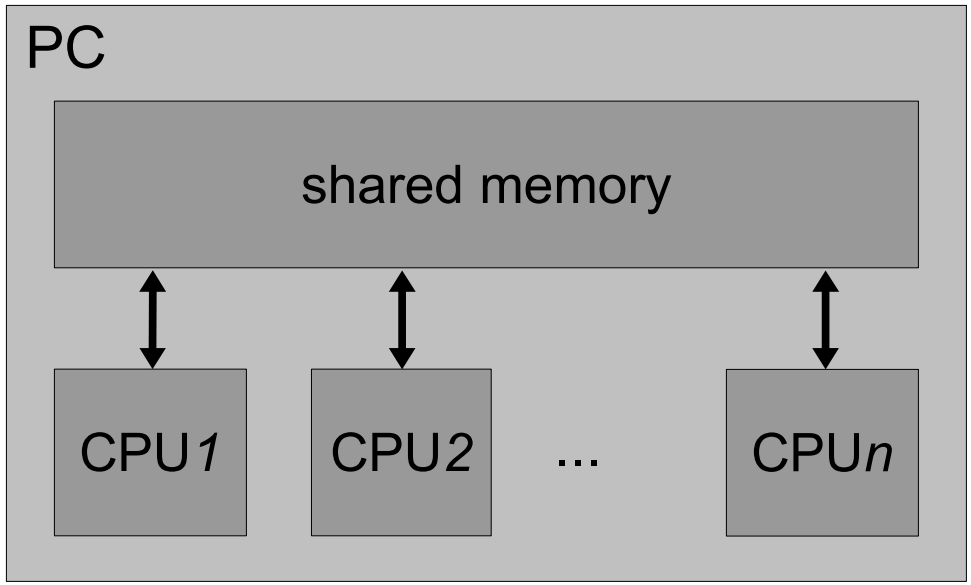
  2 – SIMD – Single Instruction Multiple Data

  3 – multi/many-cores – multi-threading

  → **mécanismes complémentaires**

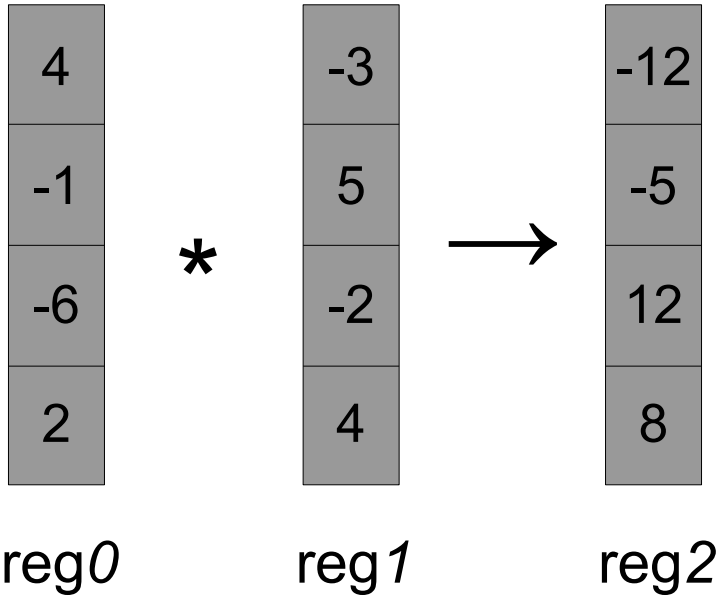
CPU - Parallélisme 1/3

- **multi/many-cores**
  - Principe : chaque cœur exécute son propre flot d'instruction (=thread)
    - thread = « processus léger »
    - un thread par cœur à la fois
    - assignation et ordonnancement par l'OS
    - les threads communiquent via la mémoire partagée
  - mise en œuvre, ex :
    - bibliothèques tierces
    - via le compilateur et OpenMP (instrumentation du code)
  - hyper-threading
    - assigner deux threads sur un même cœur
    - exécution alternée au niveau des instructions
    - permet un meilleur taux d'utilisation des unitées
    - parfois contre-productifs !

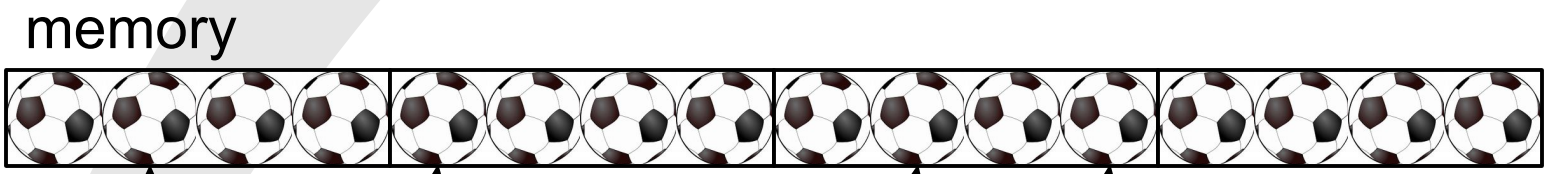


CPU - Parallélisme 2/3
SIMD

- **SIMD → Single Instruction Multiple Data**
  - Principe : exécuter la même opération sur plusieurs données en même temps
  - Jeux d'instructions vectoriels, ex :
    - SSE (x86) : registres 128 bits (4 float, 2 double, 4 int)
    - AVX (x86) : registres 256 bits (8 float, 4 double, 8 int)
    - NEON (ARM) : registres 128 bits (4 float, 4 int)
    - ...

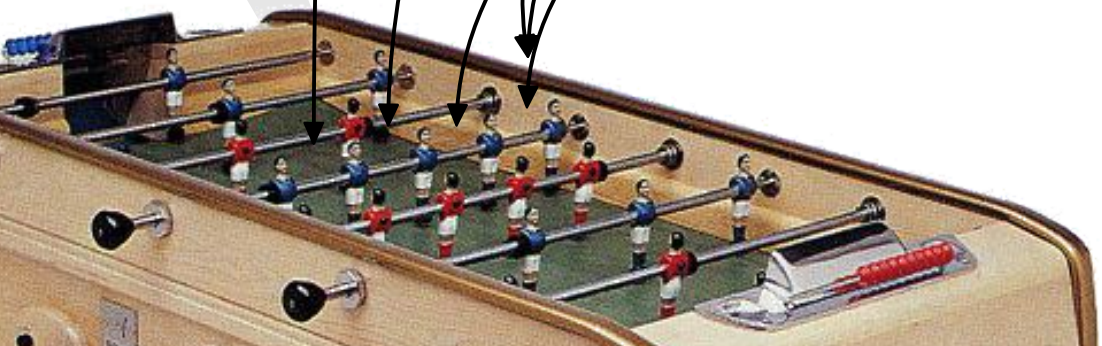


CPU - Parallélisme 2/3
SIMD



fast slower
(aligned) (not aligned)

don't be silly!



CPU - Parallélisme 2/3
SIMD

- **Exemple de stockage pour des points 3D:**
  - **Array of Structure (AoS)**

```
struct Vec3 {
  float x, y, z;
} ;
Vec3 points[100] ;
```

→ not SIMD friendly

  - **Structure of Arrays (SoA)**

```
struct Points {
  float x[100];
  float y[100];
  float z[100];
};
Points points;
```

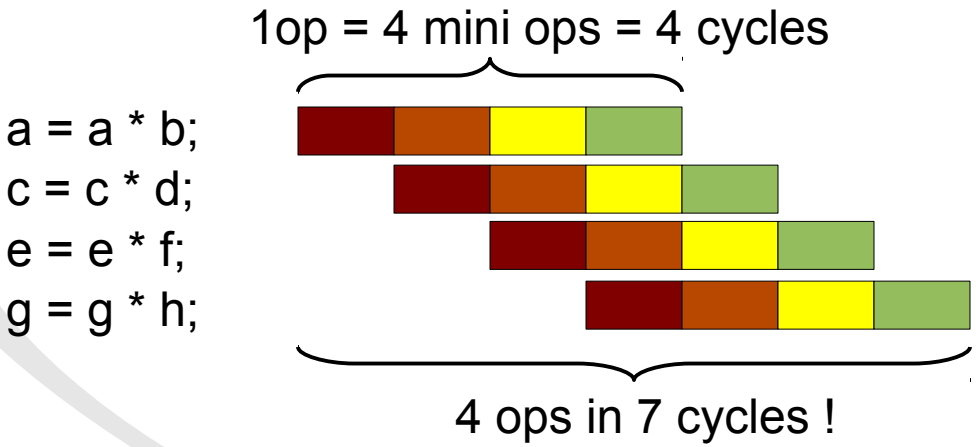
[démo]

CPU - Parallélisme 3/3

- **parallélisme au niveau des instructions**
  - **pipelining**
    - une opération = une unité de calcul (ex : addition)
    - opération décomposée en plusieurs sous-taches
      - une sous-unité par sous-tache
      - 1 cycle par sous-tache

    → plusieurs opérations peuvent s'enchainer sur une même unité

  → requière des instructions non dépendantes !



[démo]

CPU – Parallélisme 3/3
In-order / Out-of-order

- **In-order processors**
  - instruction fetch
  - attendre que les opérandes soient prêtes
    - dépendances avec les opérations précédentes ou temps d'accès mémoire/caches
  - exécuter l'instruction par l'unité respective

- **Out-of-orders processors**
  - instruction fetch
  - mettre l'instruction dans une file d'attente
  - dès que les opérandes d'une des instructions de la file d'attente sont prêtes, exécuter l'instruction par l'unité respective
  - couplée à la prédiction de branchement...

→ **réduit les temps où le processeur est en attente**

→ simplifie le travail du développeur/compilateur :)

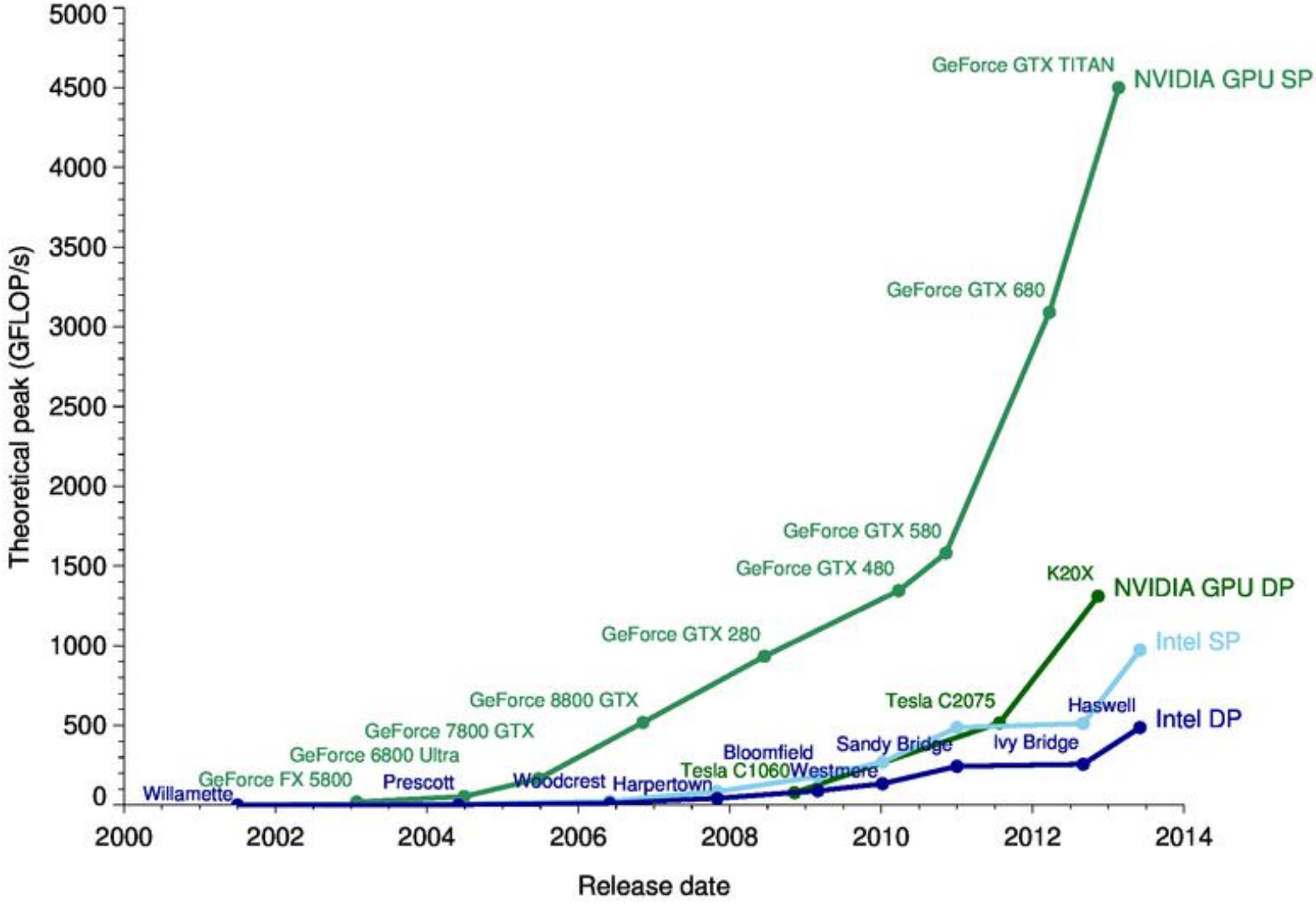
→ requière des instructions non dépendantes

Peak performance

- **Example : Intel i7 Quad CPU @** 2.6GHz **(x86_64,AVX,FMA)**
  - pipelining/OOO → 2 * (mul + add) / cycle (cas idéal)
  - AVX → x 8 **ops** simple précision à la fois
  - multi-threading → x 4
  - fréquence → x **2.6G**

  → **peak performance: 332.8 GFlops**

CPU versus GPU - Conséquence



- **GPU performant sur un certain type de calcul uniquement**

  → exploiter les deux au mieux !

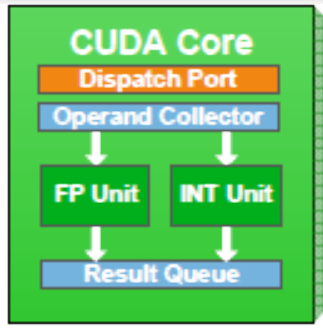
CPU versus GPU

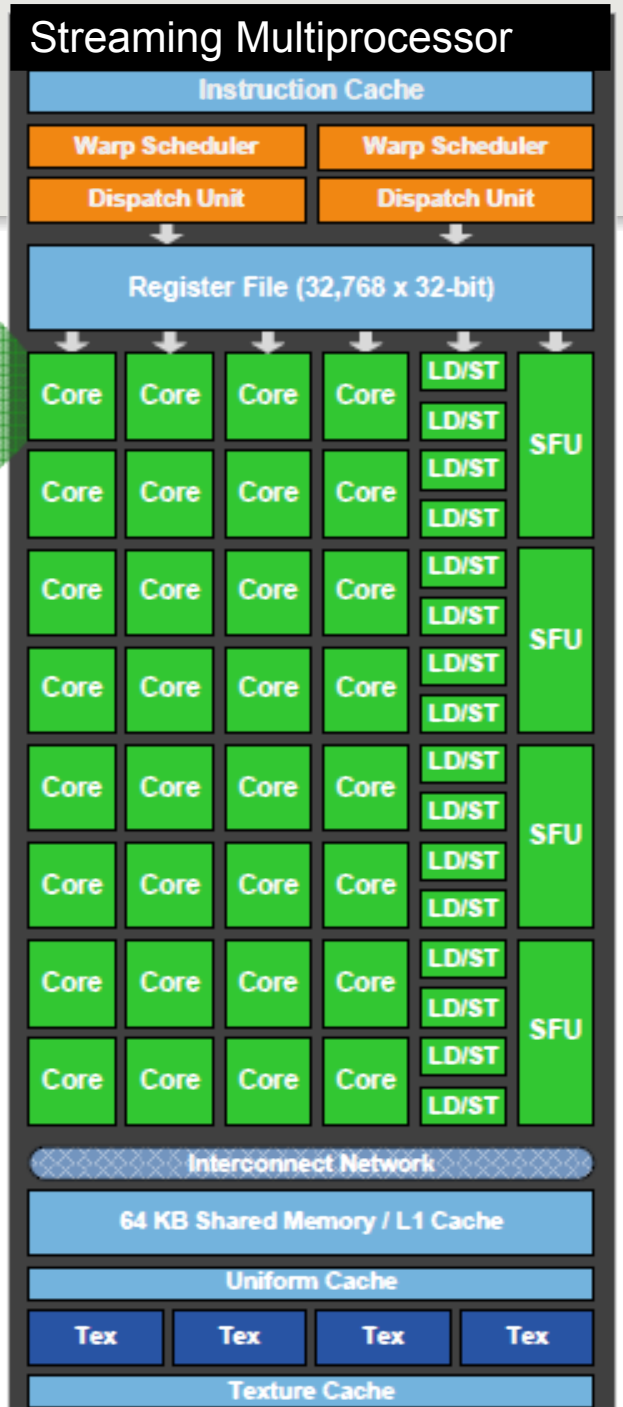
- **CPU :**
  - applications généraliste
  - optimisé pour qu'**un thread finisse le plus rapidement possible**
  - pipelining important
  - gros caches
  - logic complexe (prédiction de branchement, prefetching, out-of-order, hyper-threading)
  - parallélisation hiérarchique : thread/SIMD
- **GPU**
  - calcul flottant
  - optimisé pour qu'un **groupe** de threads finissent rapidement
  - context-switching
  - petit caches, beaucoup de registres
  - logique réduite à la gestion des threads
  - parallélisation à 1 seul niveau : threads

Architecture d'un GPU Nvidia

- Découpage hiérarchique :
  - Streaming multiprocessors regroupés par cluster
  - Chaque SM peut exécuter un programme différent (MIMD)

Architecture d'un GPU Nvidia

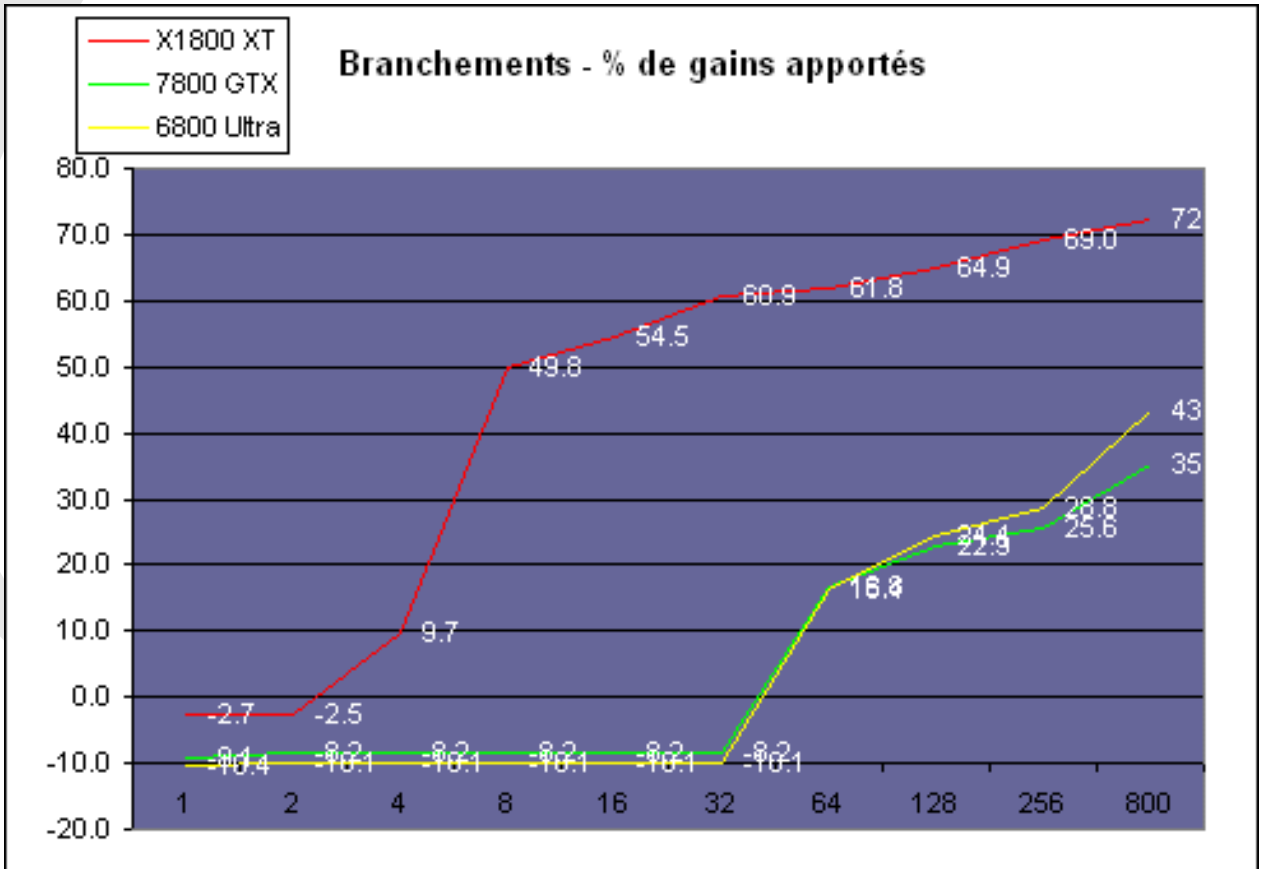




- **Streaming Multiprocessor**
  - composé de nombreuses unités de calculs
    → many-core
  - exécution synchronisée du même programme
  - approche « scalaire »
    - 1 thread par core
    - Single Program Multiple Data (SPMD)
    - Single Instruction Multiple Threads
    - !! différent du SIMD des CPU !!
  - accent sur les opérations + et * sur les float
    - sqrt, div, pow, etc. → beaucoup plus lent
    - double → (beaucoup) plus lent
  - grand banc de registres
    - limité les accès mémoire
    - context-switching

Conséquence du SPMD

- ***N* threads suivent le même chemin d'exécution**
  - durée d'exécution >= durée du thread le plus long
  - attention aux branchements dynamiques, aux boucles
- **Exemple :** branchement dynamique en fonction d'un motif à bande de 1 à 800pixels :

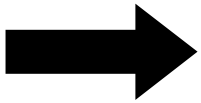


Conséquence du SPMD

- **Architecture**
  - compromis entre le nombre de cores par SM et les unités de logique/contrôle

- **Exemple**

```
for(i=0;i<k;++i) {
  if(data[i]!=0) {
     foo(i) ; // calculs complexes
  }
}
```

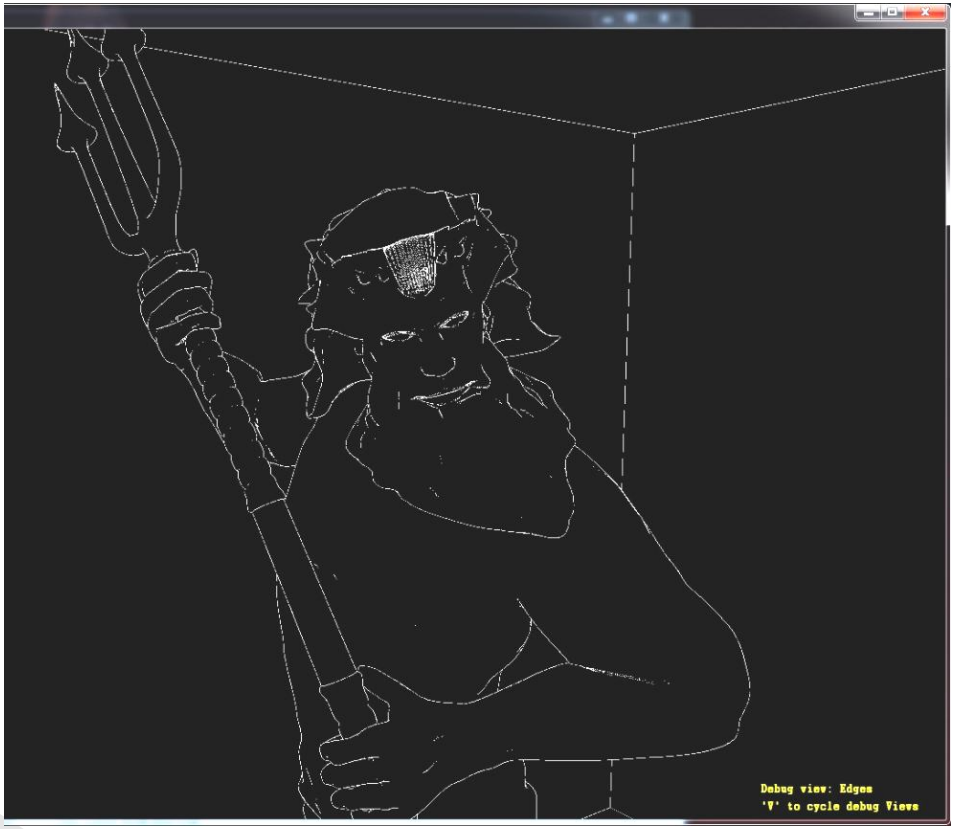


```
int i=0 ;
while(i<k) {
  while(i<k && data[i]==0) ++i;
  if(i<k) {
     foo(i) ; // calculs complexes
  }
}
```

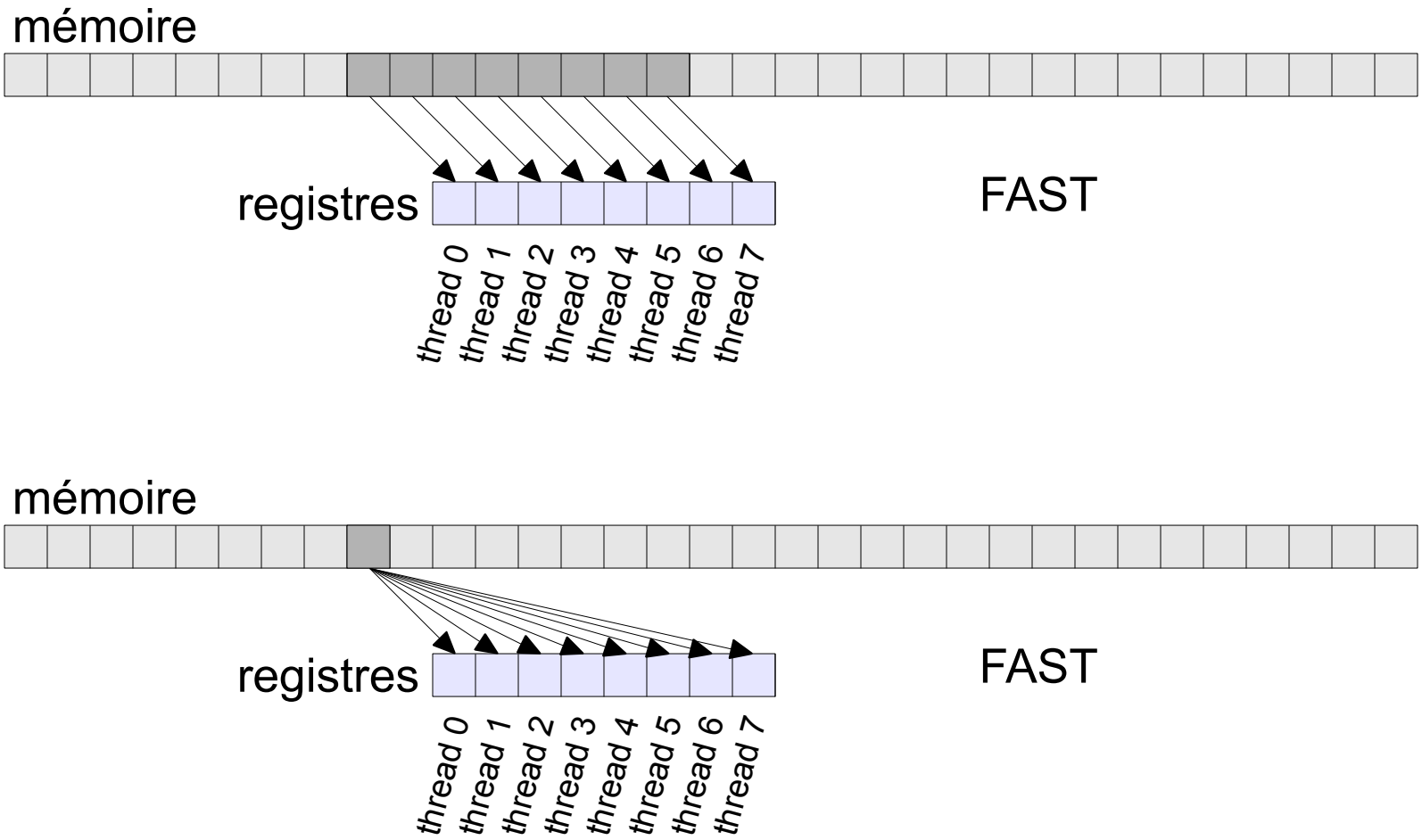
(→ de meilleures solutions existe...)

Conséquence du SPMD

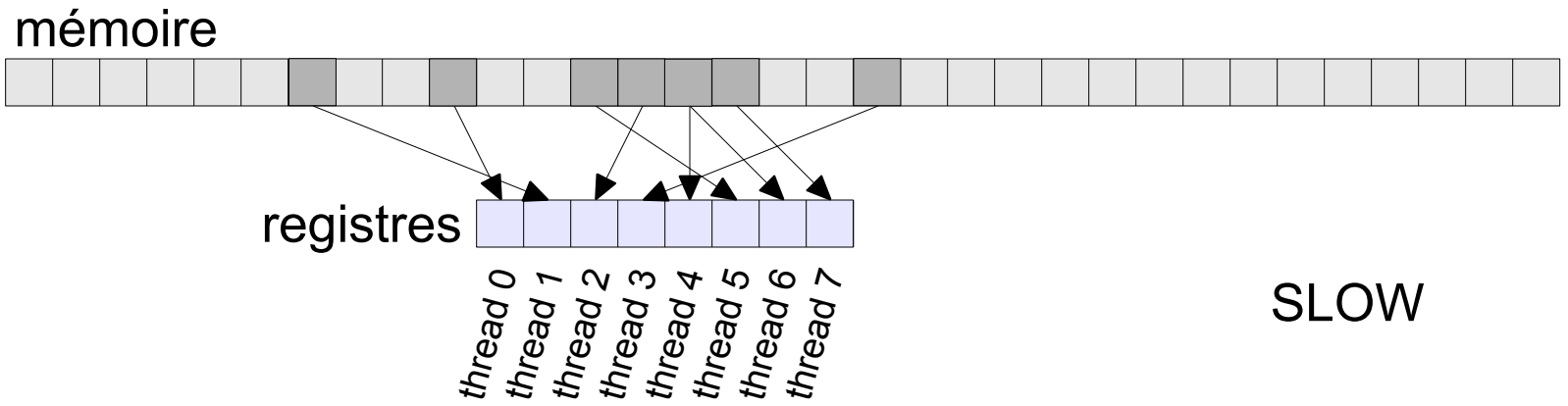
- **Exemple : traitement spécial aux silhouettes :**



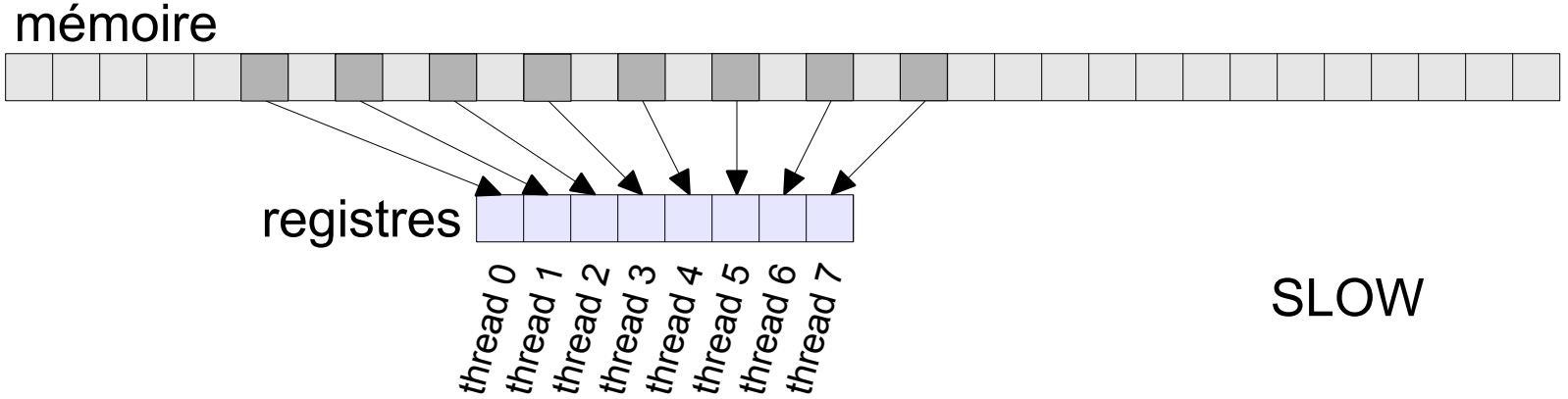
Accès mémoires → cohérence



Accès mémoires → cohérence



SLOW



SLOW

Problème des accès mémoires

- **CPU ou GPU**
  - 1 accès mémoire ↔ qques centaines de cycles !!
    (1 cycle ↔ 1 opération flottante)

- **Solutions**
  - CPU
    → caches énormes, hiérarchiques, *prefetching* auto, etc.

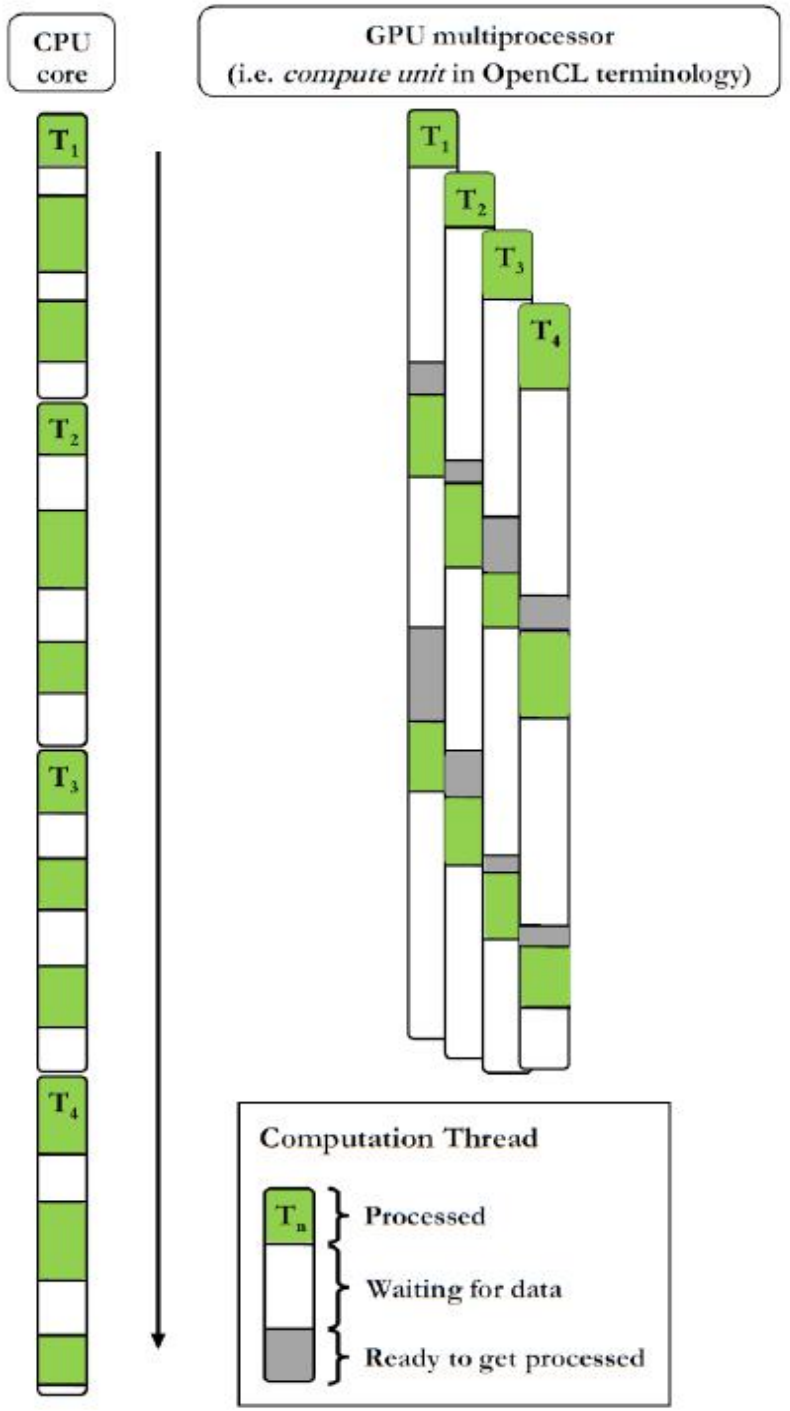
  - GPU
    - (unités de « textures » avec mini-caches supposant des accès cohérents)
    - stockage des résultats temporaires dans des registres
    - **masquage des temps de latence par mise en attente des threads**

Masquage des temps d'attentes

- **Rappel :**
  - l'objectif est d'optimiser les temps de calculs pour un « groupe » de *threads*, pas pour un *thread* unique

- **Idée :**
  - associer plusieurs paquets de *threads* à un SM
  - un seul paquet est actif à la fois
  - principe
    - si le packet actif doit attendre des données
    - alors ***context-switching*** :
      - on met le paquet en attente
      - on active un autre paquet
    - lorsque les données sont arrivées → échange
  - nécessite beaucoup de registres !

Masquage des temps de latences



Lien avec OpenGL

- **OpenGL : différent shaders exécutés en même temps (vertex, fragment, geometry, etc.)**
  - affectation dynamique d'un SM à un type de shader
  - des dizaines (centaines) de sommets (fragments) sont traités en même temps
  - vertex shader : 1 sommet ↔ 1 thread ↔ 1 core
    - *N* sommets consécutifs sont associés à un SM
      - (selon l'ordre dans lequel ils sont émis)
      - cache de sommets pour éviter de traiter le même sommet plusieurs fois (1 sommet est partagé par 6 triangles en moyenne)
  - fragment shader : 1 fragment ↔ 1 thread ↔ 1 core
    - un bloc de *N*x*M* pixels sont associés à un SM
  - geometry shader : 1 primitive ↔ 1 thread ↔ 1 core

Introduction à CUDA

gael.guennebaud@inria.fr

Comment programmer les GPU ?

- **Notion de kernel**
  - exemple (*n* produits scalaires): $c_i = \mathbf{a}_i^T \mathbf{b}$ ($\mathbf{a}_i$, $\mathbf{b}$: vecteurs 3D, c_i : scalaire)

```
for(int i=0;i<n;++i) {

    c[i] =  a[i].x * b.x
          +  a[i].y * b.y
          +  a[i].z * b.z;

}
```

} boucle sur les coefficients
→ 1 donnée = 1 thread

} **kernel**

→ **Comment définir les kernels ??**

- Architectures spécifiques
  → compilateurs spécifiques
  → langages spécifiques ?

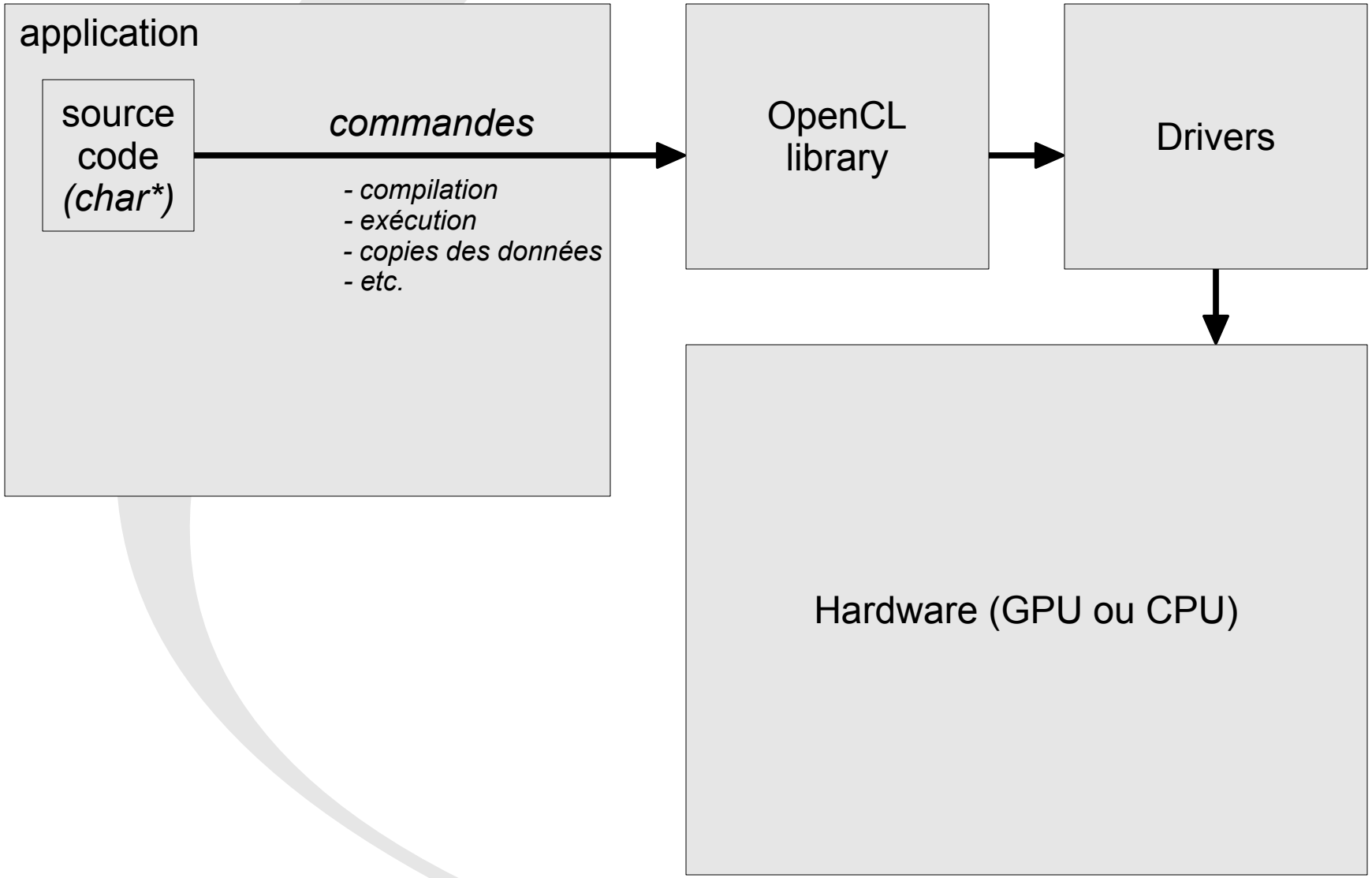
CUDA versus OpenCL

- **OpenCL**
  - Solution initiée par Apple, ouvert à la OpenGL
  - Ce veut générique (CPU/GPU)
  - Approche à la OpenGL/GLSL
  - Langage proche du C

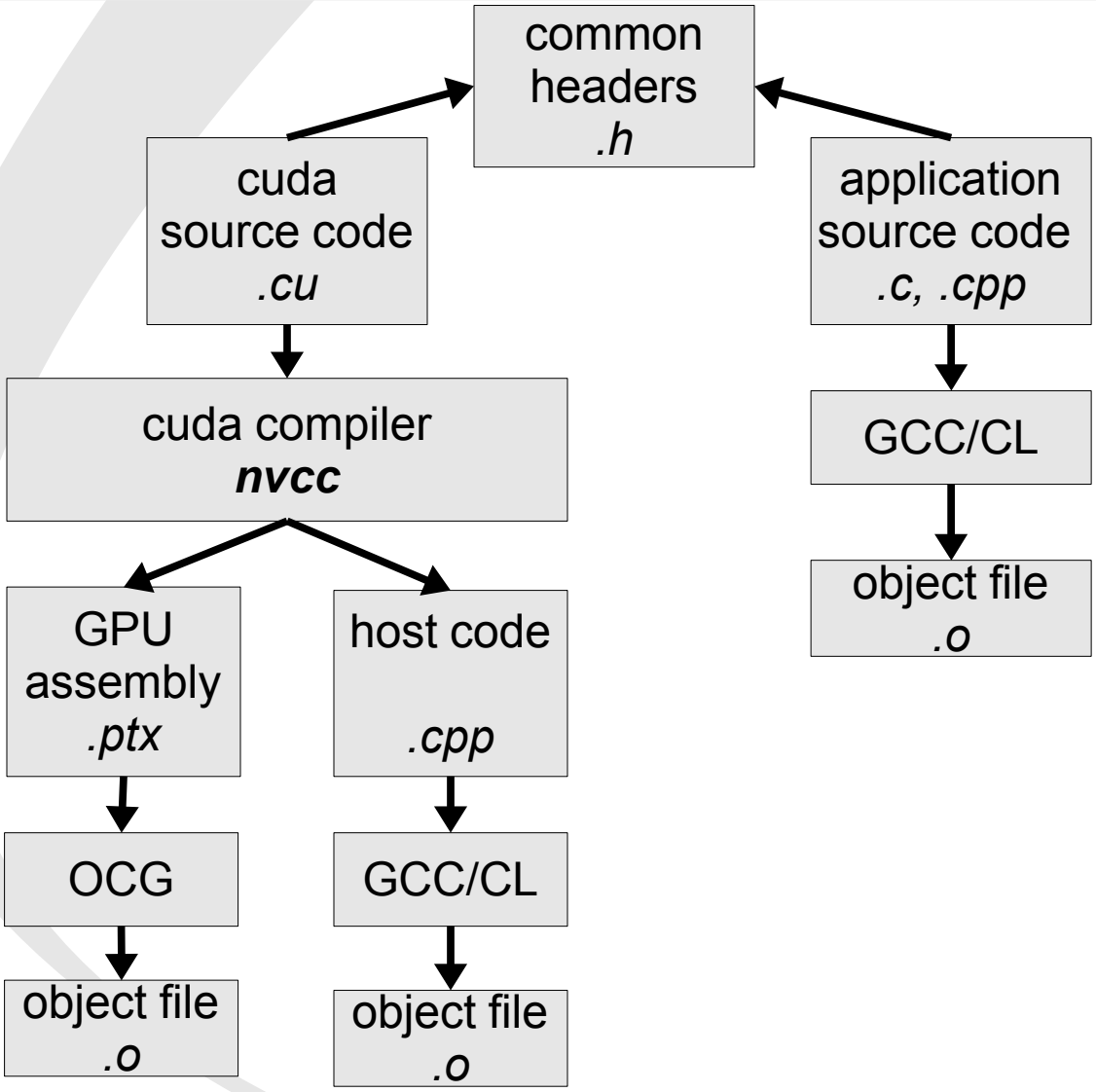
- **CUDA = « Compute Unified Device Architecture »**
  - CUDA c'est :
    - un langage (C/C++ plus extensions et restrictions)
    - un compilateur (nvcc) + drivers
    - interfaces pour Fortran, python, MatLab, Java, etc.
    - des bibliothèques : CUBLAS, Thrust, etc.
  - Supporte uniquement le matériel Nvidia (GPU et Tesla)
    - bientôt support par AMD !

- **OpenACC ↔ OpenMP**
  - bientôt dans GCC

OpenCL


application
source code (char*)
commandes
- compilation
- exécution
- copies des données
- etc.
OpenCL library
Drivers
Hardware (GPU ou CPU)

CUDA C/C++


common
headers
.h
cuda
source code
.cu
application
source code
.c, .cpp
cuda compiler
nvcc
GCC/CL
object file
.o
GPU
assembly
.ptx
host code
.cpp
OCG
GCC/CL
object file
.o
object file
.o

CUDA : qques notions de bases

- **Compute device**
  - (co)processeur avec DRAM et nombreux threads en parallèle
  - typiquement → GPU

- **Device kernel**
  - fonction principale exécutée sur un *compute device* par de nombreux cœurs
  - préfixé par __global__ :

```
__global__ void my_kernel(...) {
  …
}
```

Types de fonctions

- **Différent types de fonctions :**

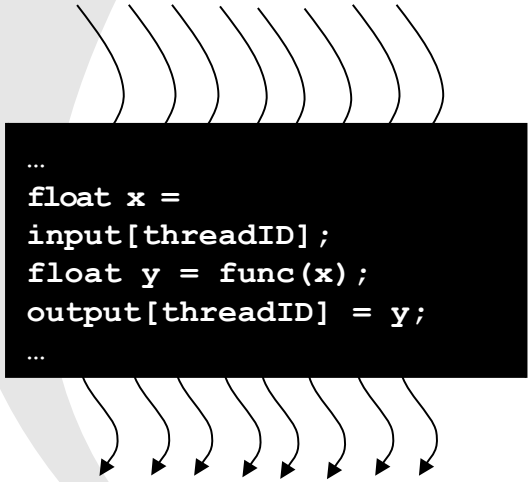
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc();`	device	device
`__global__ void KernelFunc();`	device	host
`__host__ float HostFunc();`	host	host

  - On peut avoir __host__ et __device__ en même temps
  - __global__ = kernel, retourne void (similaire aux main() des shaders)
  - __device__ : pas de variable static local à la fonction, pas de nombre variable d'arguments

CUDA : qques notions de bases

- **Paradigme : tableau de threads** *(1D, 2D, ou 3D)*
  - chaque thread a son propre numéro (ID) utilisé pour accéder aux données et savoir quoi faire

threadID

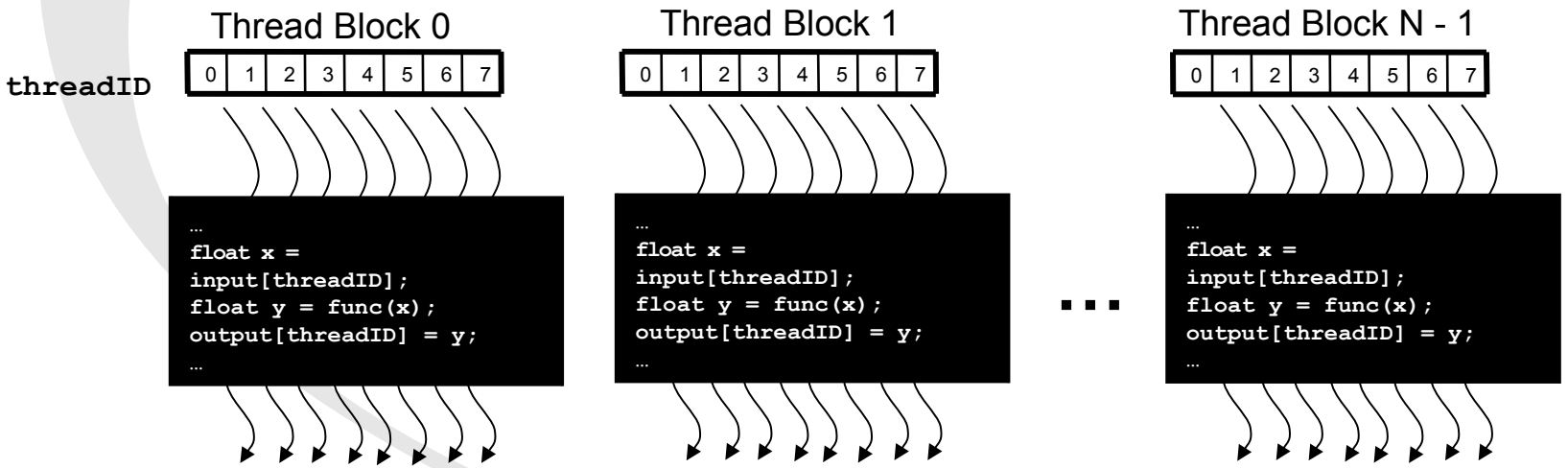



```
struct Vec3 { float x, y, z ; } ;

__global__ void my_kernel(  const Vec3* a,
                            Vec3 b,
                            float* c)
{
   int i = threadID;

   c[i] =  a[i].x * b.x
         +  a[i].y * b.y
         +  a[i].z * b.z;
}
```

Host : générer *n* threads exécutant my_kernel(a, b, c) ;

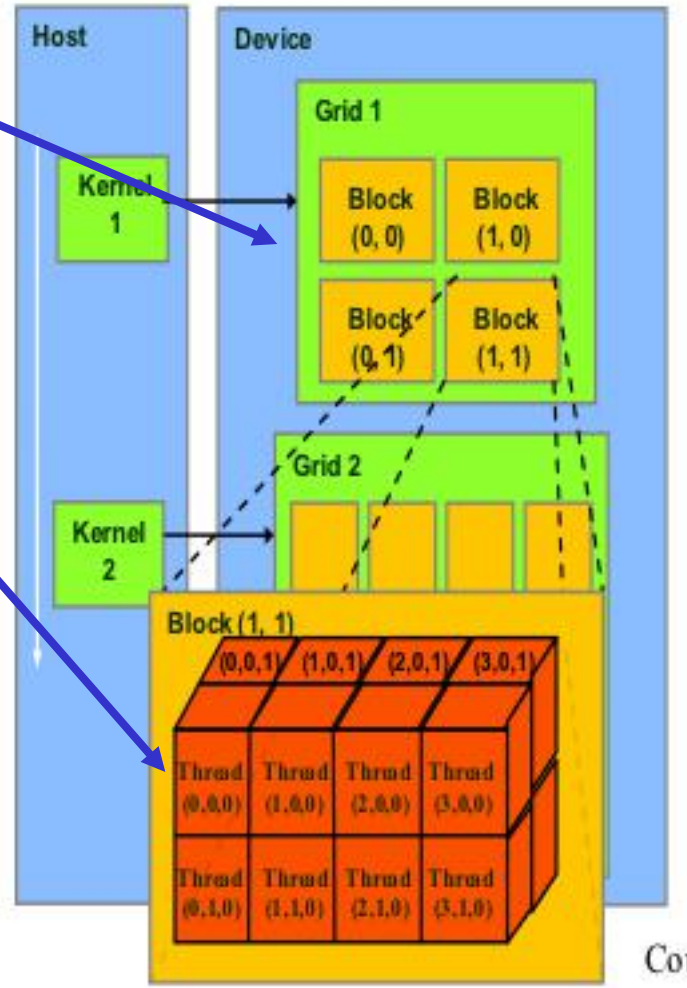
Thread Blocks

- **Les threads sont regroupés en blocs**
  - Les threads d'un même bloc peuvent coopérer :
    - mémoire partagée, opérations atomiques, synchronisation (**`barrière`**)
    - ils sont exécutés sur un même SM
    - nb threads/bloc >> nb cores d'un SM
  - Les threads de deux blocs différents ne peuvent pas communiquer !
    - les blocs sont exécutés dans un ordre arbitraire
      - plusieurs blocs peuvent être assignés à un même SM
      - exécuté de manière indépendante sur plusieurs SM



Block ID et Thread ID

- **threadIdx = numéro du thread dans le bloc courant (3D)**
- **blockIdx =numéro du bloc dans la grille (2D)**
- **Global thread ID =**
  - Block ID (1D ou 2D)
  - + Thread ID (1D, 2D, 3D)
- **Choix du nombre de threads / bloc par l'utilisateur**
- **nD → simplifie les accès mémoires pour les données multi-dimensionelles**
  - Traitement d'image
  - Image volumique
  - équations différentielles



Courtesy

Appeler un kernel cuda

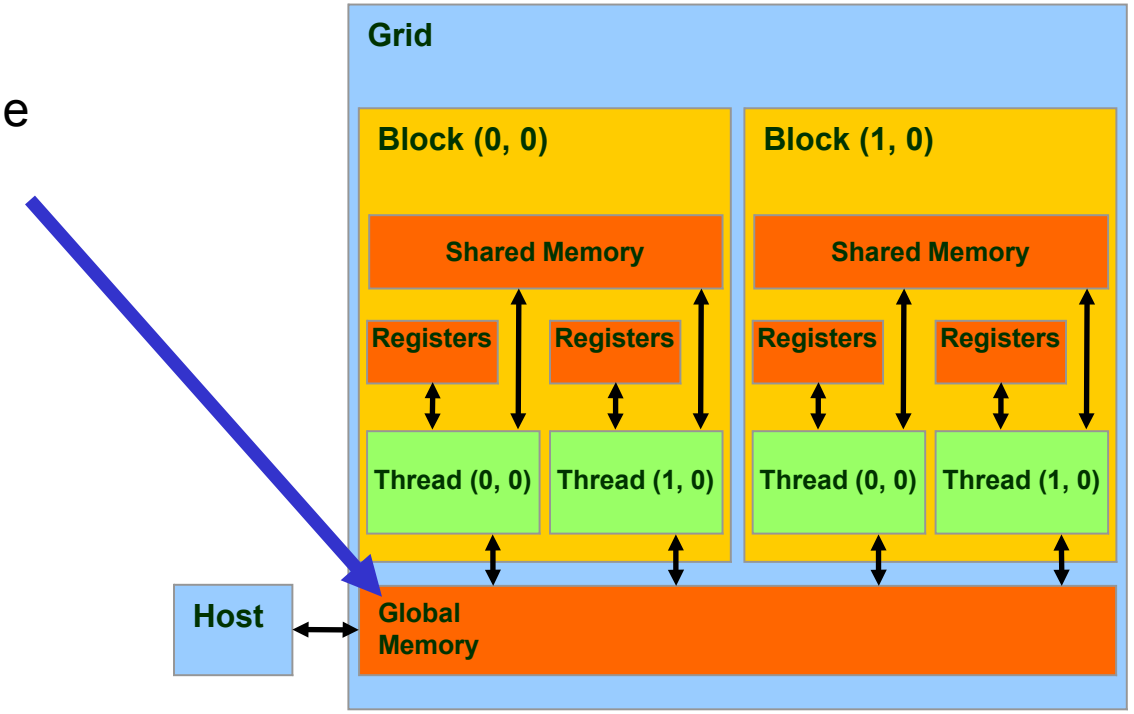
- **Un kernel est appelé avec une configuration d'exécution**
  - appelé dans dans un fichier .cu :

```
__global__ void my_kernel(...) ;
…
dim3 DimGrid(100,50) ; // 100*50*1 = 5000 blocks
dim3 DimBlock(4,8,8) ; // 4*8*8 = 256 threads / blocks

my_kernel<<< DimGrid, DimBlock >>> (…) ;
```

  - exécution asynchrone

Gestion de la mémoire

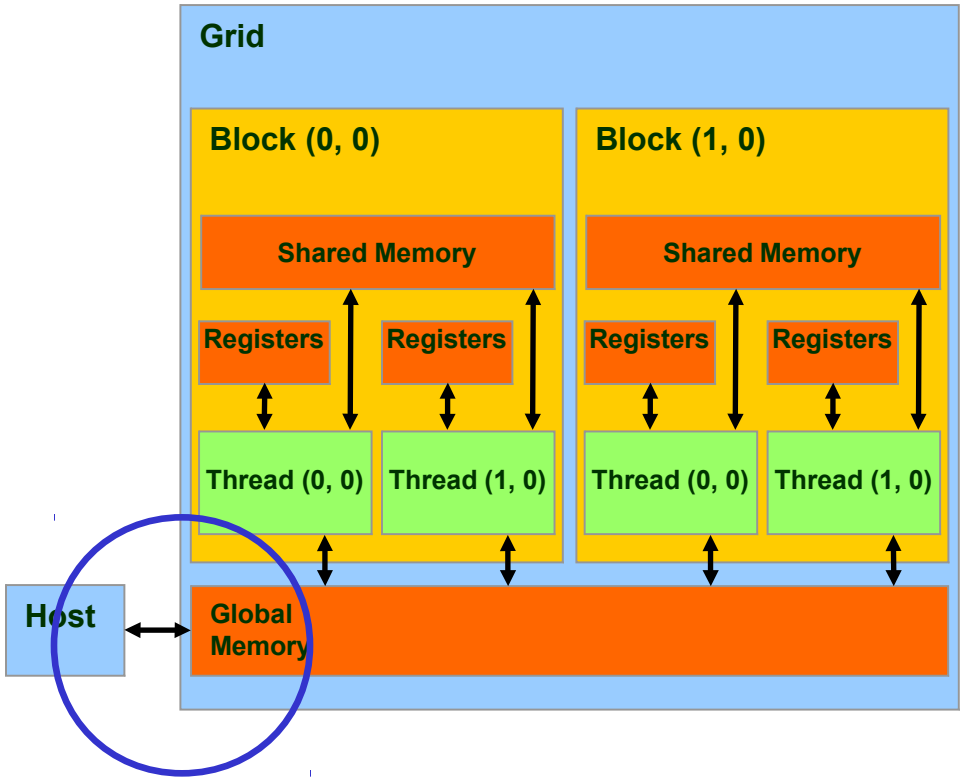
- **cudaMalloc()**
  - alloue de la mémoire globale sur le GPU
  - libérée avec cudaFree()




```
int n = 64 ;
float* d_data ; // convention d_ pour device
int size = 64*64*sizeof(float)
cudaMalloc((void**)&d_data, size) ;
...
cudaFree(d_data) ;
```

Transfert de données

- **cudaMemcpy()**
  - transferts asynchrones
  - 4 paramètres
    - ptr de destination
    - ptr de source
    - nombre d'octets
    - type : **cudaMemcpy***
      - ...HostToHost
      - ...HostToDevice
      - ...DeviceToDevice
      - ...DeviceToHost




```
float* h_data = malloc(sizeof(float)*n*n) ;
h_data = … ;
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice) ;
...
cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost) ;
```

Exemple I

- **Calculer *n* produits scalaires**

```
__global__ void many_dots_kernel(const Vec3* a, Vec3 b, float* c) {
    int i = threadIdx.x;
    c[i] =  a[i].x * b.x   +   a[i].y * b.y   +   a[i].z * b.z;
}

__host__ void many_dots(const std::vector<Vec3>& a, const Vec3& b, std::vector<float>& c) {
    Vec3* d_a; // d_ for device
    float* d_c;
    int n = a.size() ;
    cudaMalloc((void**)&d_a, n*sizeof(Vec3));
    cudaMalloc((void**)&d_c, n*sizeof(float));

    cudaMemcpy(d_a, &a[0].x, n*sizeof(Vec3), cudaMemcpyHostToDevice) ;

    dim3 DimGrid(1,1);
     dim3 DimBlock(n,1);
    many_dots_kernel<<< DimGrid, DimBlock >>> (d_a, b, d_c);

    cudaMemcpy(&c[0], d_c, n*sizeof(float), cudaMemcpyDeviceToHost) ;
    cudaFree(d_a);
    cudaFree(d_c) ;
}
```

!! un problème !!

Exemple I

- **Calculer *n* produits scalaires**

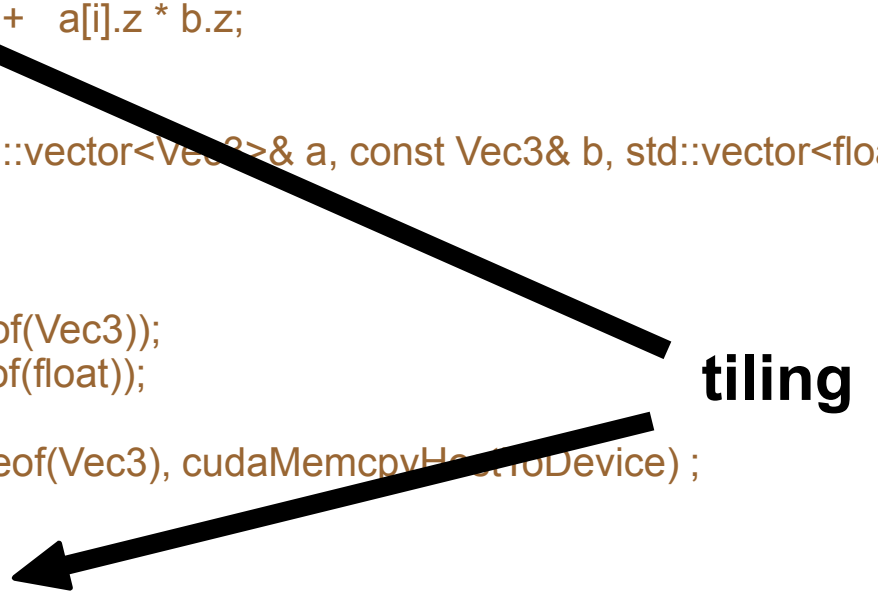
```
__global__ void many_dots_kernel(const Vec3* a, Vec3 b, float* c) {
    int i = threadIdx.x;
    c[i] =  a[i].x * b.x   +   a[i].y * b.y   +   a[i].z * b.z;
}

__host__ void many_dots(const std::vector<Vec3>& a, const Vec3& b, std::vector<float>& c) {
    Vec3* d_a; // d_ for device
    float* d_c;
    int n = a.size() ;
    cudaMalloc((void**)&d_a, n*sizeof(Vec3));
    cudaMalloc((void**)&d_c, n*sizeof(float));

    cudaMemcpy(d_a, &a[0].x, n*sizeof(Vec3), cudaMemcpyHostToDevice) ;

    dim3 DimGrid(1,1);
     dim3 DimBlock(n,1);
    many_dots_kernel<<< DimGrid, DimBlock >>> (d_a, b, d_c);

    cudaMemcpy(&c[0], d_c, n*sizeof(float), cudaMemcpyDeviceToHost) ;
    cudaFree(d_a);
    cudaFree(d_c) ;
}
```

!! un seul bloc → un seul SM sera utilisé

Exemple I

- **Calculer *n* produits scalaires**

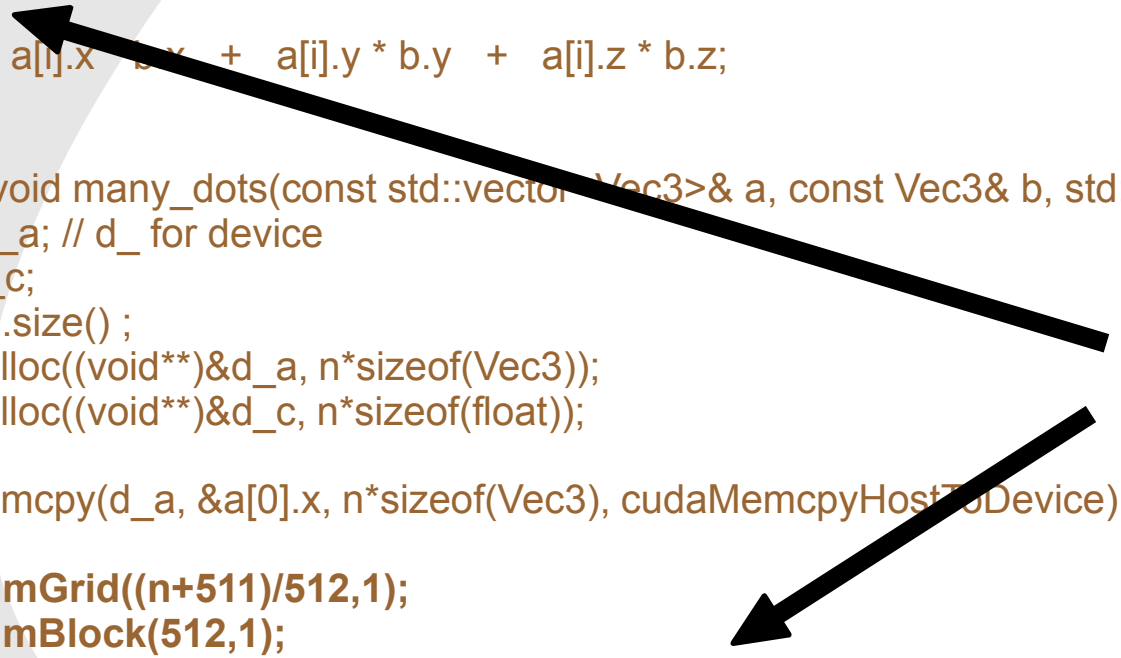
```
__global__ void many_dots_kernel(const Vec3* a, Vec3 b, float* c) {
    int i = blockId.x * blockDim.x  +  threadIdx.x;
    c[i] =  a[i].x * b.x   +   a[i].y * b.y   +   a[i].z * b.z;
}

__host__ void many_dots(const std::vector<Vec3>& a, const Vec3& b, std::vector<float>& c) {
    Vec3* d_a; // d_ for device
    float* d_c;
    int n = a.size() ;
    cudaMalloc((void**)&d_a, n*sizeof(Vec3));
    cudaMalloc((void**)&d_c, n*sizeof(float));

    cudaMemcpy(d_a, &a[0].x, n*sizeof(Vec3), cudaMemcpyHostToDevice) ;

    dim3 DimGrid((n+511)/512,1);
     dim3 DimBlock(512,1);
    many_dots_kernel<<< DimGrid, DimBlock >>> (d_a, b, d_c);

    cudaMemcpy(&c[0], d_c, n*sizeof(float), cudaMemcpyDeviceToHost) ;
    cudaFree(d_a);
    cudaFree(d_c) ;
}
```



Exemple I

- **Calculer *n* produits scalaires**

```
__global__ void many_dots_kernel(int n const Vec3* a, Vec3 b, float* c) {
    int i = blockId.x * blockDim.x  +  threadIdx.x;
    if(i<n)
        c[i] =  a[i].x * b.x  +  a[i].y * b.y  +  a[i].z * b.z;
}

__host__ void many_dots(const std::vector<Vec3>& a, const Vec3& b, std::vector<float>& c) {
    Vec3* d_a; // d_ for device
    float* d_c;
    int n = a.size() ;
    cudaMalloc((void**)&d_a, n*sizeof(Vec3));
    cudaMalloc((void**)&d_c, n*sizeof(float));

    cudaMemcpy(d_a, &a[0].x, n*sizeof(Vec3), cudaMemcpyHostToDevice) ;

    dim3 DimGrid((n+511)/512,1);
    dim3 DimBlock(512,1);
    many_dots_kernel<<< DimGrid, DimBlock >>> (n, d_a, b, d_c);

    cudaMemcpy(&c[0], d_c, n*sizeof(float), cudaMemcpyDeviceToHost) ;
    cudaFree(d_a);
    cudaFree(d_c) ;
}
```



! out of range !

Choisir la taille des blocs/grilles

- **Choisir la taille des blocs/grilles**
  - warp = ensemble de threads exécutés en même temps sur un même SM
    - taille = f(nombre de cores)
    - ex : 32 threads / warp

      → **taille des blocs multiple de 32**
  - 1 seul warp actif à la fois par SM, mais échange entre les warps actifs et en pause afin de masquer les attentes

    → objectif : **maximiser le nombre de warp / SM**
  - si grand nombre de threads
    → optimiser la taille des blocs (voir exemple)
    → en déduire les dimensions de la grille
  - sinon réduire la taille des blocs pour occuper tous les SM
    - nombre maximal de threads par bloc < nb_total_threads / nb_SM
    - nombres de blocs = multiple du nombre de SMs

→ **benchmark automatique pour optimiser les dimensions**

Choisir la taille des blocs/grilles

- **Exemple de GPU: G80**
  - caractéristiques :
    - 32 threads / warp
    - 8 blocs max par SM
    - 768 threads max par SM
    - 15 SMs
  - block sizes:
    - 8x8 → 64 threads → x8 → 512 threads < 768 → non optimal
    - 16x16 → 256 threads → x3 → 768 threads → optimal ?
    - 32x32 → 1024 threads → x1 → 1024 > 768 → impossible

32x8 → pour la cohérence des accès mémoires

- **En réalité**
  - plus complexe car dépend également de la complexité du kernel :
    - nombres de registres limités
      → nombre de threads/SM peut être largement inférieur

Comment paralléliser son code sur GPU ?

- CPU : 4-16 threads à la fois
- GPU : plusieurs milliers !

Comment paralléliser son code ?

- **Dépend de nombreux facteurs**
  - Nature du problème
  - Dimensions du problème
  - Hardware visé

  → **Pas de solution universelle**

  → **Pas de solution automatique (compilateur)**
- **Acquérir de l'expérience**
  - étudier des cas « simples »
  - retrouver des motifs connus au sein d'applications complexes
- **Dans tous les cas**
  - 1 - identifier les taches indépendantes → approche hiérarchique
  - 2 - regrouper les taches → trouver la bonne granularité

Défis

- **Trouver et exploiter la concurrence**
  → regarder le problème sous un autre angle
  - *Computational thinking* (J. Wing)
  - nombreuses façons de découper un même problème en taches
- **Identifier et gérer les dépendances**
  - dépend du découpage
- **De nombreux facteurs influent sur les performances**
  - surcoût du au traitement parallèle
  - charge non équilibrée entre les unités de calculs
  - partage des données inefficace
  - saturation des ressources (bande-passante mémoire)
  - ...

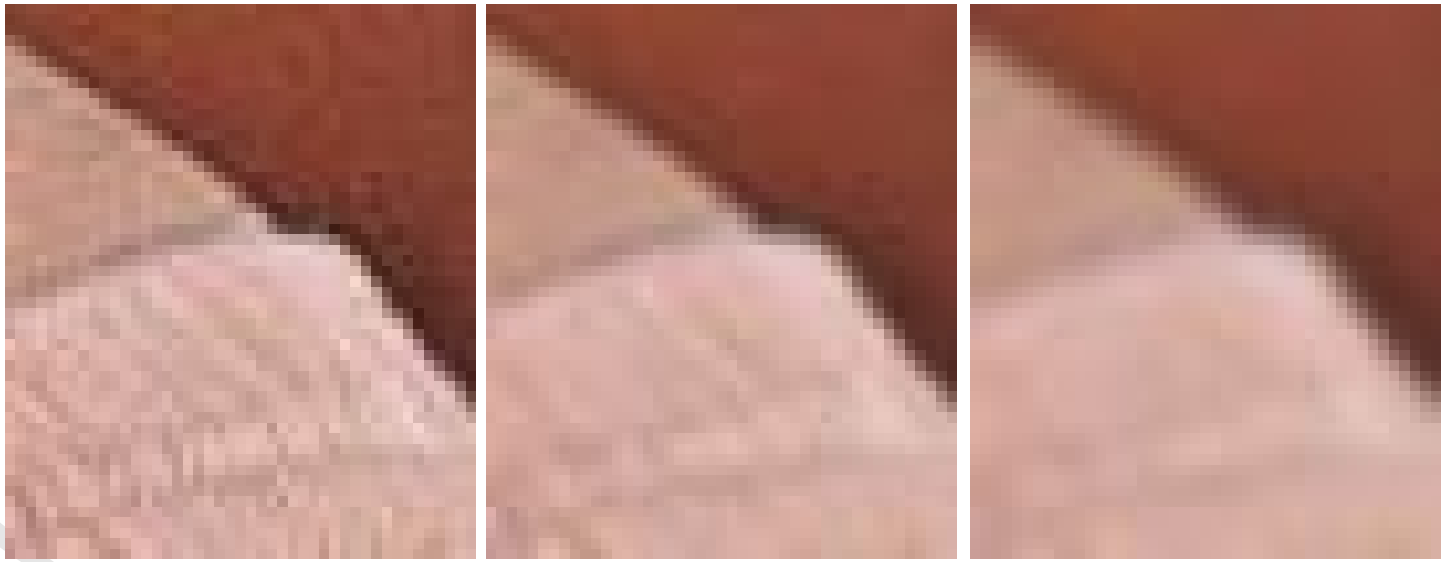
Exemple II - Convolution

- **Convolution discrète**
  - Application d'un filtre *f* de rayon *r* :

$$I[x,y] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} I[x+i, y+j] * f[i,j]$$

  - Exemple de filtre Gaussien 3x3 :
    (filtre basse-bas, supprime les hautes fréquences, lissage/flou)

$$f = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$



<démo> input 1 passe 3 passes

Exemple II

- **Appliquer un filtre à une image**

```
__global__ void apply_filter(int w, int h, float* img, float filter[3][3]) {
  int x = threadIdx.x ;
  int y = threadIdx.y ;
  if(x<w && y<h) {
```

$$\text{img}[x,y] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} \text{img}[x+i, y+j] * \text{filter}[i,j];$$

```
  }
}
```

!! deux problèmes !!

```
__host__ void apply_filter(MyImage& image, float filter[3][3]) {
  float* d_img;

  /* … */

  dim3 DimGrid(1,1) ;
  dim3 DimBlock(image.width(),image.height(),1) ;
  apply_filter<<< DimGrid, DimBlock >>> (image.width(), image.height(), d_img, filter) ;

  /* … */
}
```

Exemple II

- **Appliquer un filtre à une image**

```
__global__ void apply_filter(int w, int h, float* img, float filter[3][3]) {
  int x = threadIdx.x ;
  int y = threadIdx.y ;
  if(x<w && y<h) {
```

$$\text{img}[x,y] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} \text{img}[x+i, y+j] * \text{filter}[i,j];$$

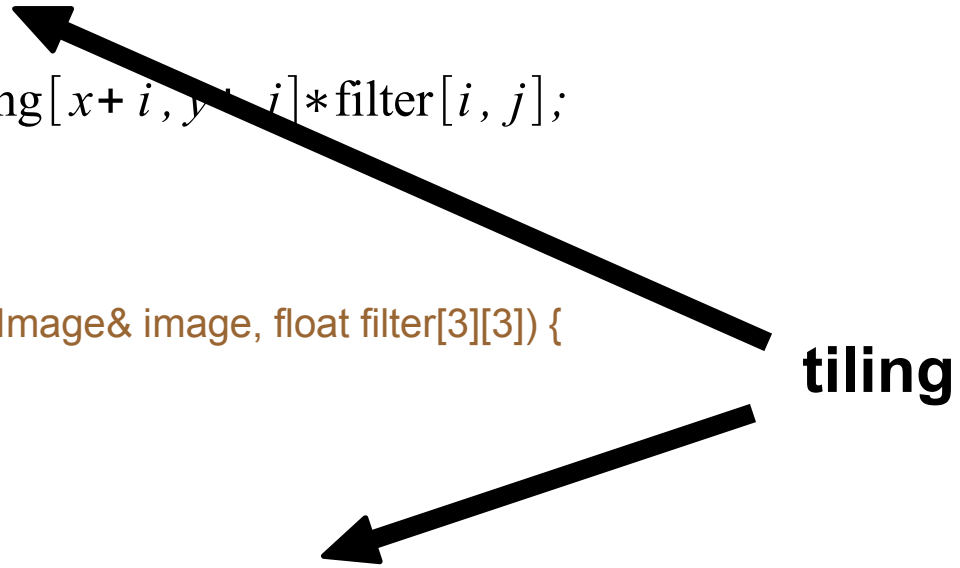
```
  }
}

__host__ void apply_filter(MyImage& image, float filter[3][3]) {
  float* d_img;

  /* … */

  dim3 DimGrid(1,1) ;
  dim3 DimBlock(image.width(),image.height(),1) ;
  apply_filter<<< DimGrid, DimBlock >>> (image.width(), image.height(), d_img, filter) ;

  /* … */
}
```

!! un seul bloc → un seul SM sera utilisé

Exemple II

- **Appliquer un filtre à une image**

```
__global__ void apply_filter(int w, int h, const float* img, float filter[3][3]) {
  int x = blockIdx.x*blockDim.x + threadIdx.x ;
  int y = blockIdx.y*blockDim.y + threadIdx.y ;
  if(x<w && y<h) {
    img[x,y]= Σ_{i=-r}^{r} Σ_{j=-r}^{r} img[x+i,y+j]*filter[i,j];
  }
}

__host__ void apply_filter(MyImage& image, float filter[3][3]) {
  float *d_img;

  /* … */

  dim3 DimBlock(16,16,1) ;
  dim3 DimGrid((image.width()+15)/16, (image.height()+15)/16) ;
  apply_filter<<< DimGrid, DimBlock >>> (image.width(), image.height(), d_img, filter) ;

  /* … */
}
```



Exemple II

- **Appliquer un filtre à une image**

```
__global__ void apply_filter(int w, int h, const float* img, float filter[3][3]) {
  int x = blockIdx.x*blockDim.x + threadIdx.x ;
  int y = blockIdx.y*blockDim.y + threadIdx.y ;
  if(x<w && y<h) {
```

$$\mathbf{img}[x,y] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} \mathbf{img}[x+i, y+j] * \mathrm{filter}[i,j];$$

```
  }
}

__host__ void apply_filter(MyImage& image, float filter[3][3]) {
  float *d_img;

  /* … */

  dim3 DimBlock(16,16,1) ;
  dim3 DimGrid((image.width()+15)/16, (image.height()+15)/16) ;
  apply_filter<<< DimGrid, DimBlock >>> (image.width(), image.height(), d_img, filter) ;

  /* … */
}
```

!! lecture-écriture dans le même buffer avec aliasing → résultat incorrect

Exemple II

- **Appliquer un filtre à une image**

```
__global__ void apply_filter(int w, int h, const float* input, float* output, float filter[3][3]) {
  int x = threadIdx.x ;
  int y = threadIdx.y ;
  if(x<w && y<h) {
```

$$\mathbf{output}[x,y] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} \mathbf{input}[x+i, y+j] * \mathrm{filter}[i,j];$$

```
  }
}

__host__ void apply_filter(MyImage& image, float filter[3][3]) {
  float *d_input, *d_output;

  /* … */

  dim3 DimBlock(16,16,1) ;
  dim3 DimGrid((image.width()+15)/16, (image.height()+15)/16) ;
  apply_filter<<< DimGrid, DimBlock >>> (image.width(), image.height(), d_input, d_output, filter) ;

  /* … */
}
```

→ **utiliser deux buffers !**

N-body simulation

- **Exercice**
  - on a *N* particules avec :
    - masses : M*i*
    - vitesse : V*i*
    - position : P*i*
  - forces d'attractions entre 2 particules *i* et *j* :

$$F_{i,j} = \frac{g M_i M_j}{\|P_i - P_j\|^2}$$

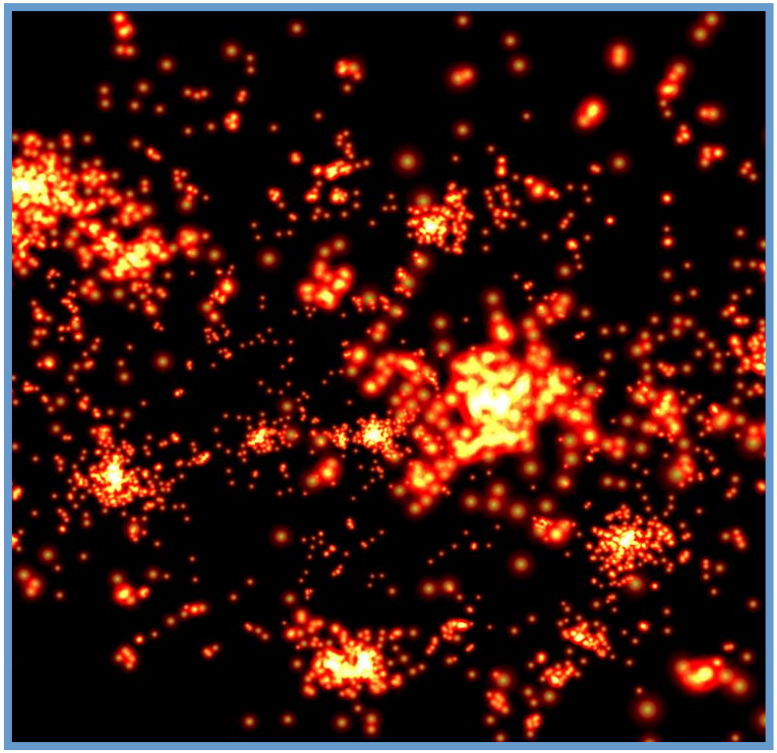
  - forces appliquées à une particule *i* :

$$F_i = \sum_{j \neq i} F_{i,j}$$

  - on en déduit les nouvelles vitesses et nouvelles positions

$$v_i(t + \Delta t) = v_i(t) + F_i \Delta t$$
$$p_i(t + \Delta t) = p_i(t) + v_i(t) \Delta t$$

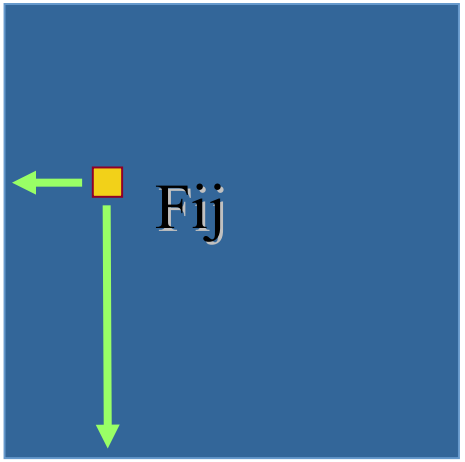
  - exemple : 5k particules



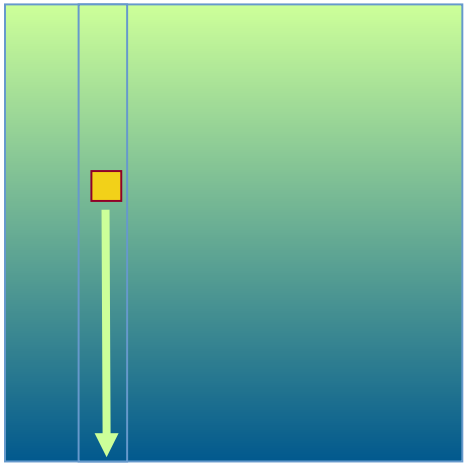
$g = 6.7 \cdot 10^{-11} N\, m^2\, kg^{-2}$

N-body simulation - Solution 1

- **étape 1**
  - calculer les Fij dans un tableau 2D
  - kernel = calcul de 1 Fij
  - nombre de threads = N^2



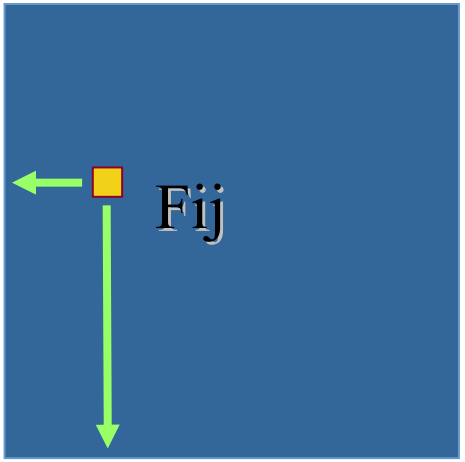
- **étape 2**
  - calcul des $F_i = \sum_{j \neq i} F_{i,j}$
    et mise à jour des vitesses et positions
  - kernel = calcul de 1 Fi
  - nombre de threads = N



- **Limitations**
  - coût mémoire : N^2
  - peu de parallélisme (N threads)

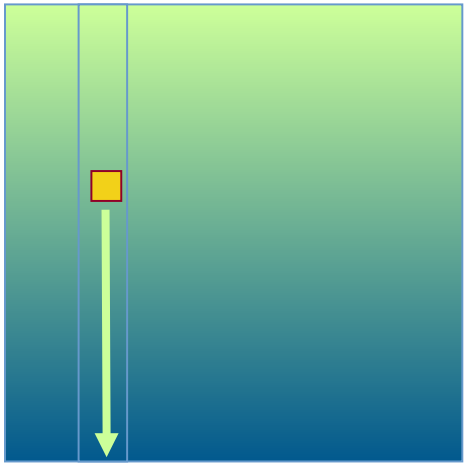
N-body simulation - Solution 2

- **étape 1**
  - calculer les Fij dans un tableau 2D
  - kernel = calcul 1 Fij
  - nombre de threads = N^2



- **étape 2**
  - réduction parallèle (somme) dans une direction ????

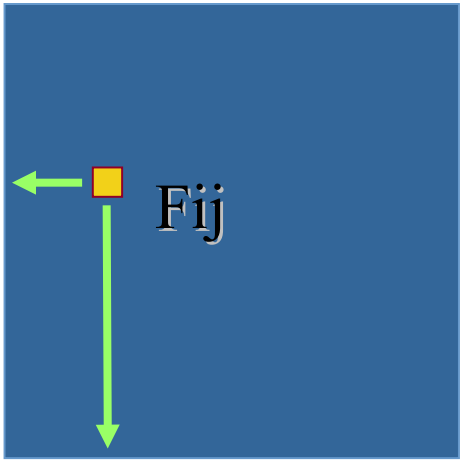
- **étape 3**
  - mise à jour des vitesses et positions



- **Limitations**
  - coût mémoire : N^2

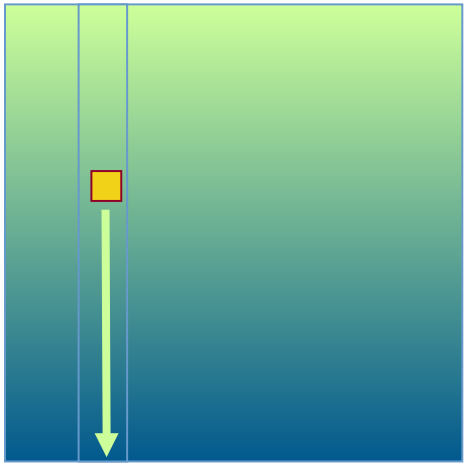
N-body simulation – Solution 2

- **étape 1 - [MAP]**
  - calculer les Fij dans un tableau 2D
  - kernel = calcul 1 Fij
  - nombre de threads = N^2



- **étape 2 - [REDUCE]**
  - réduction parallèle (somme) dans une direction ????

- **étape 3 - [MAP]**
  - mise à jour des vitesses et positions



- **Limitations**
  - coût mémoire : N^2

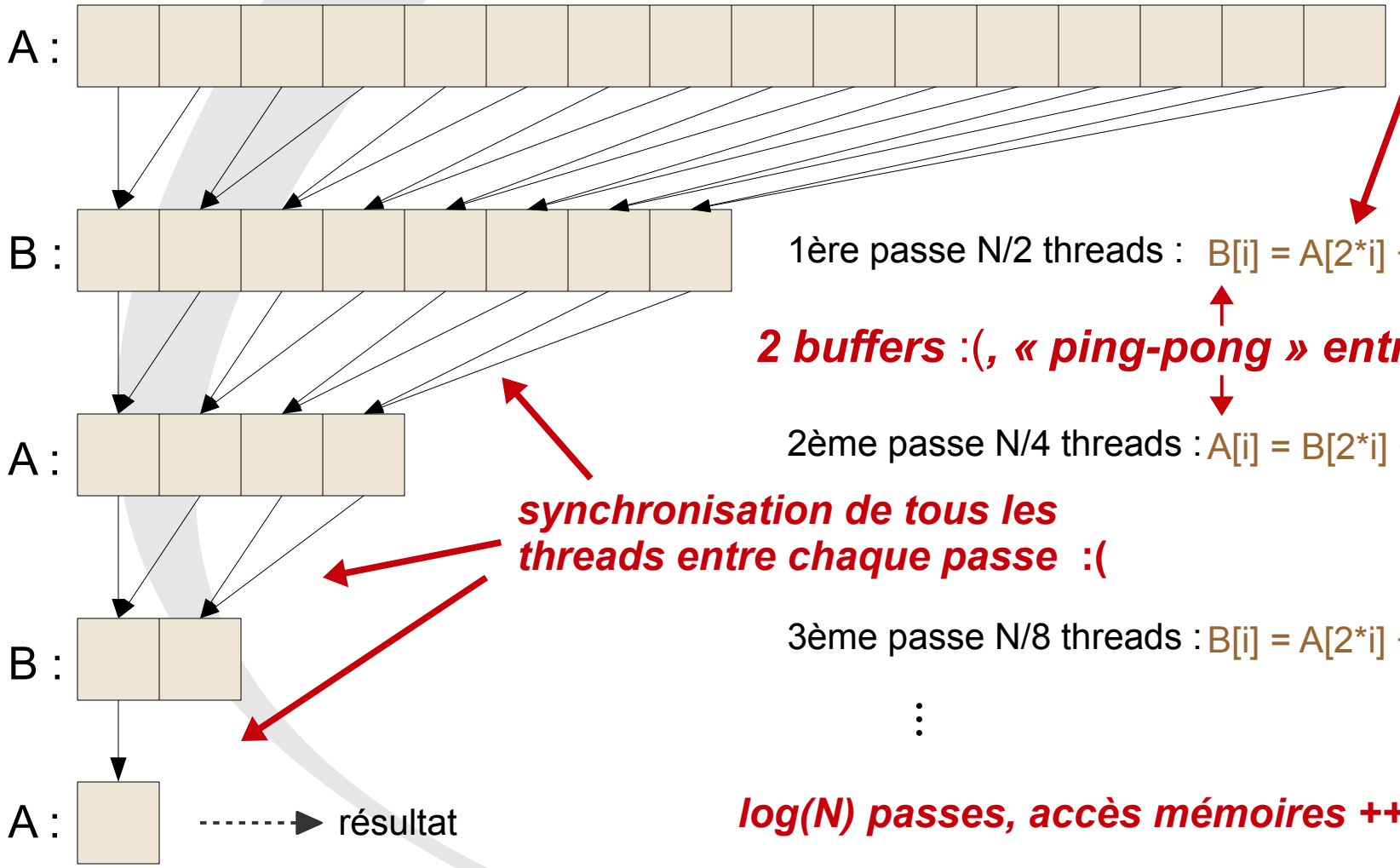
Map & Reduce

- **Premières opérations de bases :**
  - **Map =** appliquer la même fonction à chacun des éléments d'un tableau
    - Parallélisme évident :
      - 1 thread ↔ 1 élément
  - **Reduce =** réduire un tableau (ou sous-tableau) à une seule valeur
    - Exemples : somme, produit, min, max, etc.
    - Comment réaliser cette opération avec un maximum de parallélisme ?

      → exercice !

Réduction

- **Réduction 2 à 2, 1ère version :**



accès mémoires par wrap non séquentiel :(

1ère passe N/2 threads : B[i] = A[2*i] + A[2*i+1] ;

2 buffers :(, « ping-pong » entre A et B

2ème passe N/4 threads : A[i] = B[2*i] + B[2*i+1] ;

synchronisation de tous les threads entre chaque passe :(

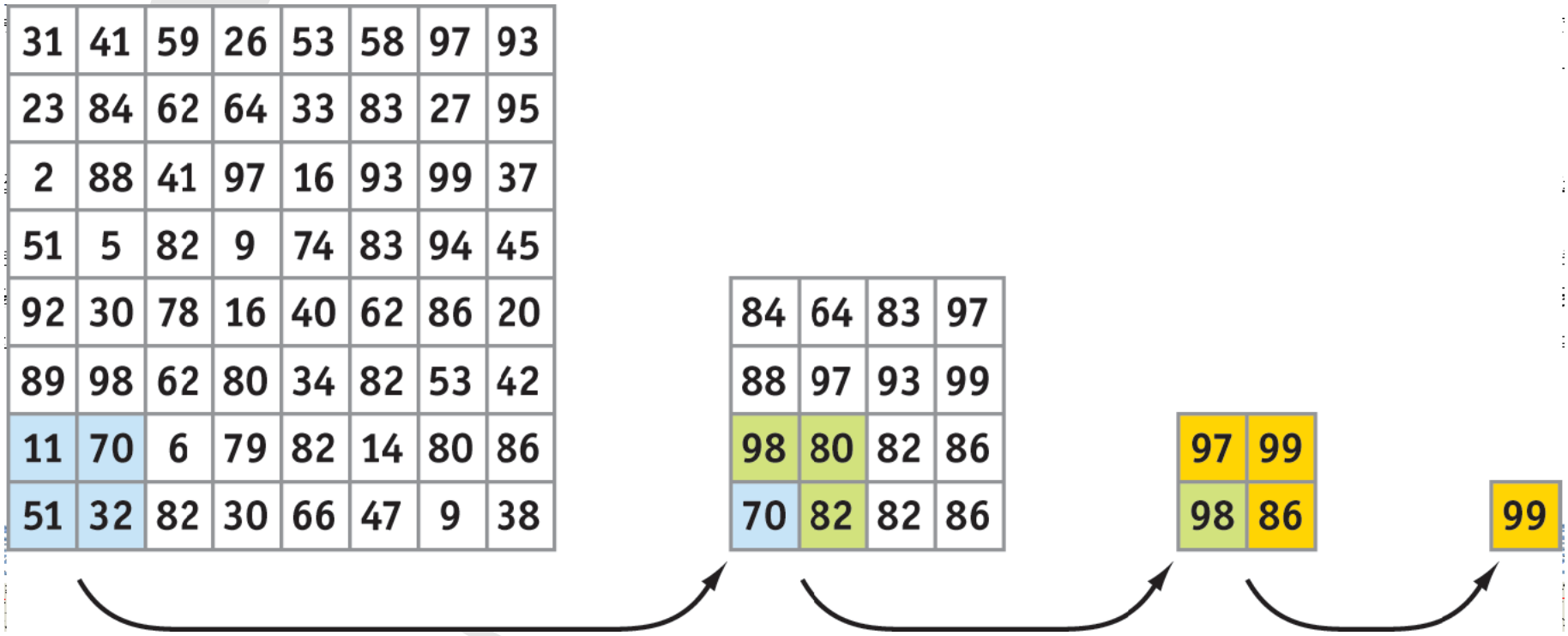
3ème passe N/8 threads : B[i] = A[2*i] + A[2*i+1] ;

⋮

log(N) passes, accès mémoires ++, :(

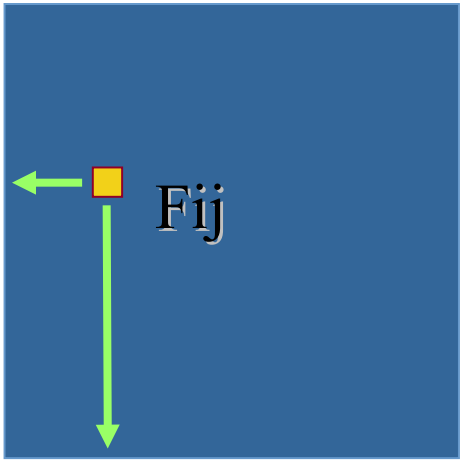
Réduction

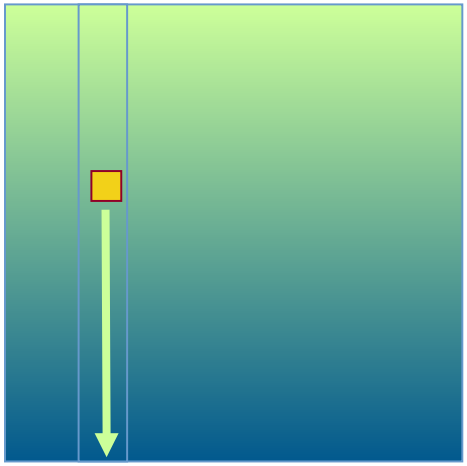
- **Exercice : calcul du min/max**
  - algo récursif, n/2 threads
  - synchronisation entre les passes → log(n) passes
  - différentes stratégies : inplace ↔ ping-pong ; accès mémoires



N-body simulation – Solution 2

- **étape 1**
  - calculer les Fij dans un tableau 2D
  - kernel = calcul 1 Fij
  - nombre de threads = N^2
- **étape 2**
  - réduction parallèle (somme) dans une direction → Fi (tableau 1D)
  - log2(N) passes
  - kernel = F(i,j)+F(i,j+1)
  - nb threads : N/2, N/4, N/8, N/16, ...
- **étape 3**
  - mise à jour des vitesses et positions
- **Limitations**
  - coût mémoire : N^2
  - kernels très (trop?) simples





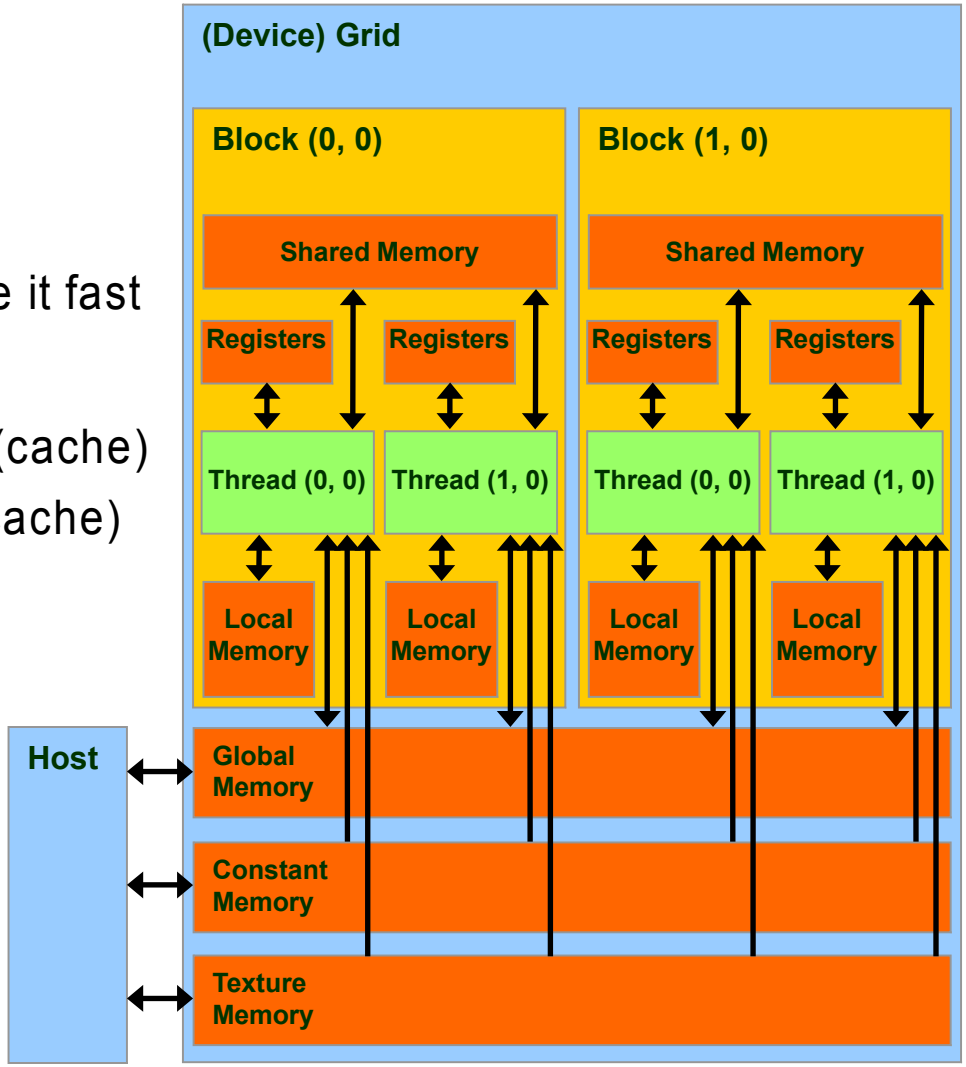
CUDA et son espace mémoire

shared-memory

CUDA Device Memory Space

- **Each thread can:**
  - R/W per-thread registers – very fast
  - R/W per-thread local memory – fast
  - R/W per-block shared memory – fast
    - 16kB/SM, can be difficult to make it fast
  - R/W per-grid global memory – very slow
  - Read only per-grid constant memory – fast (cache)
  - Read only per-grid texture memory – fast (cache)

The host can R/W global, constant, and texture memories



CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
`__device__ __local__`	`int LocalVar;`	local	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

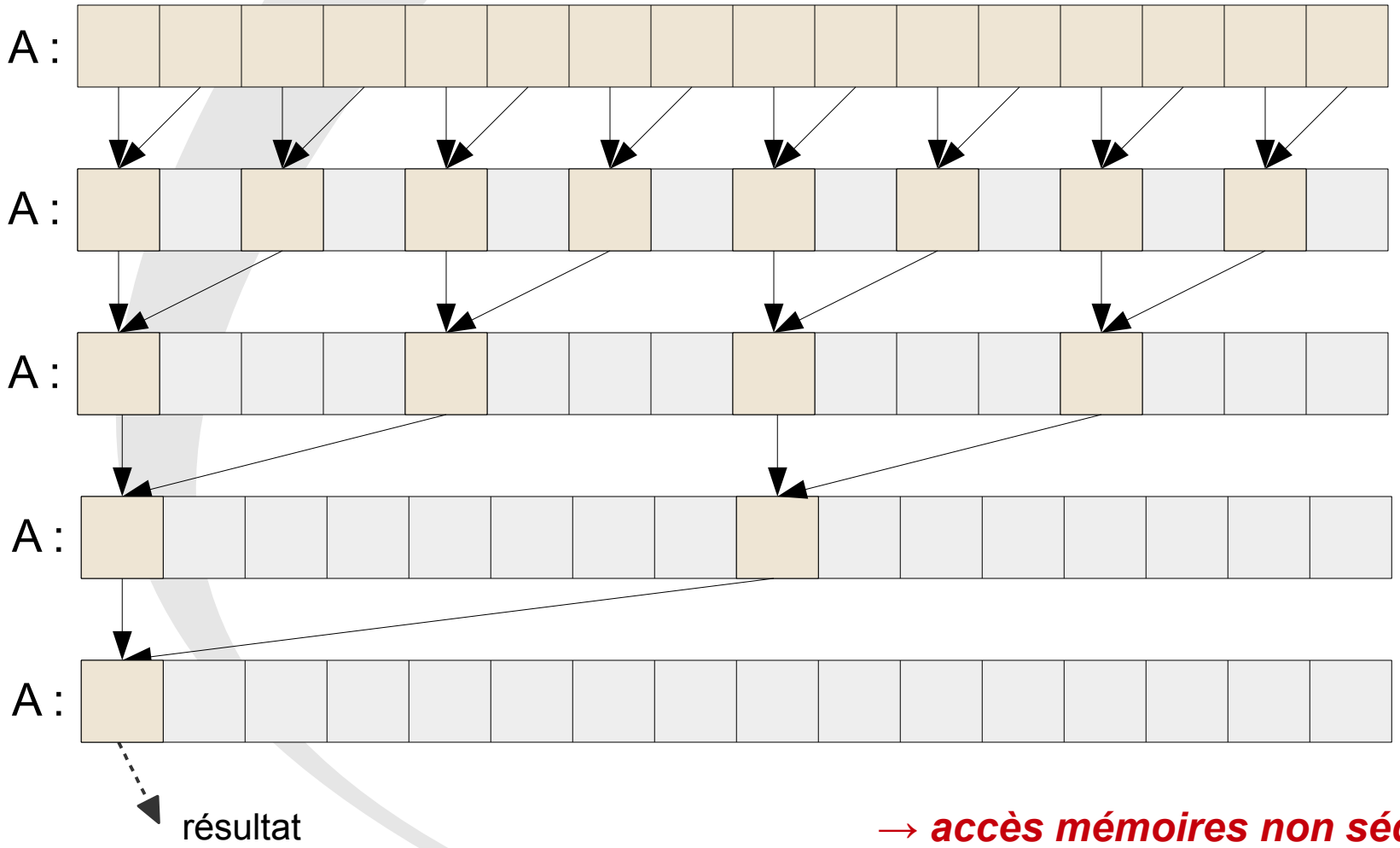
- **`__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`**
- **Automatic variables without any qualifier reside in a register**
  - Except arrays that reside in local memory

Shared Memory Ex : Réduction revisitée

- **Réduction classique :**
  - log(N) passes...
  - démarrer une nouvelle passe est coûteux
  - nombreux accès mémoires redondants
  - synchronisation **globales** des threads
- **Idée**
  - Copier une partie du tableau en shared memory
    - 2048 threads/bloc → 4k floats accumulés / bloc (SM=16kB → OK)
    - divise par 4k la taille du vecteur à chaque passe (au lieu de /2)
    - synchronisation au sein d'un bloc uniquement
    - **accès à la mémoire globale réduits**
  - Réduction partielle « in-place » en utilisant exclusivement la *shared memory*
    - !! adopter la bonne stratégie pour réduire les divergences

Shared Memory Ex : Réduction revisitée

- **Réduction 2 à 2, in-place, version 1 :**



→ ***accès mémoires non séquentiels***
→ ***divergence de branche***

Shared Memory Ex : Réduction revisitée

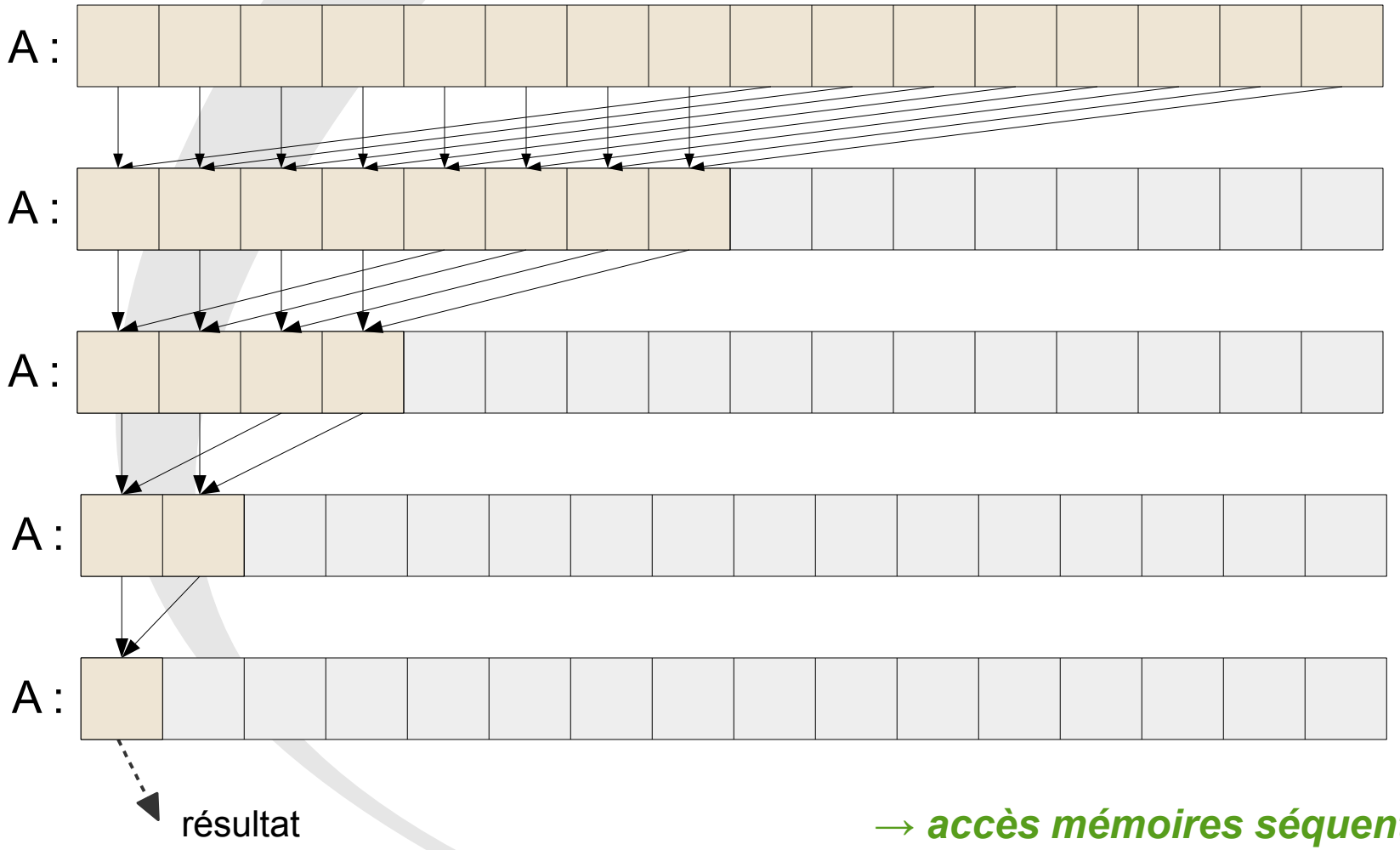
```
__shared__ float partialSum[N1];
int i      = 2*threadIdx.x;
int offset = 2*blockIdx.x*blockDim.x;

partialSum[i]   = src[offset + i];
partialSum[i+1] = src[offset + i+1];

for(int stride=1; stride<=blockDim.x ; stride*=2)
{
  __syncthreads();
  if( (i%(2*stride)==0) && (i+stride<2*blockDim.x))
    partialSum[i] = partialSum[i] + partialSum[i+stride];
}
if(i==0)
  dst[blockIdx.x] = partialSum[0];
```

accès mémoires non séquentiels

divergence de branche
→ de nombreuses unités de calcul sont mises en attente sans être libérées

Shared Memory Ex : Réduction revisitée

- **Réduction 2 à 2, in-place, version 2 :**



→ *accès mémoires séquentiels*
→ *pas de divergence de branche*

Shared Memory Ex : Réduction revisitée

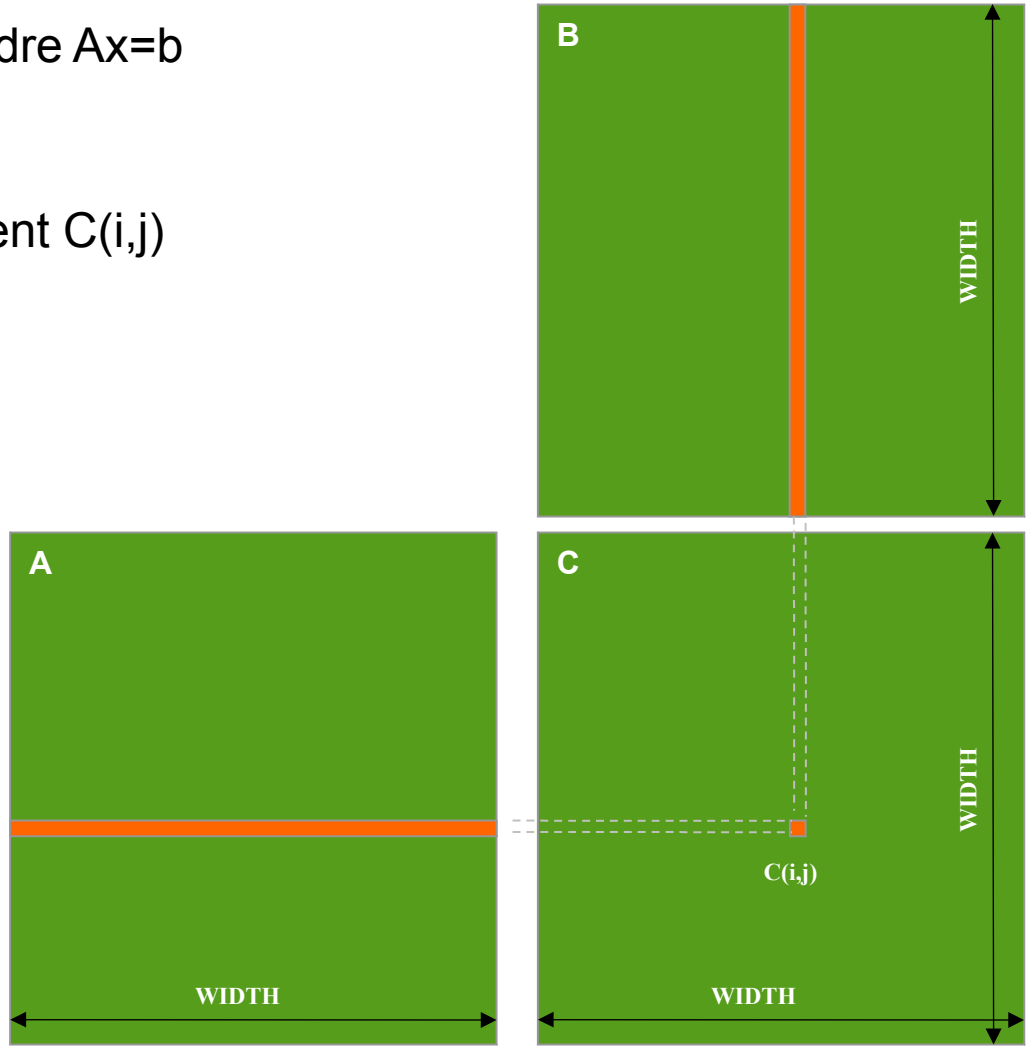
```
int i = threadIdx.x;
int offset = 2*blockIdx.x*blockDim.x;
__shared__ float partialSum[N1] ;
partialSum[i]                  = data[offset+i];
partialSum[i+blockDim.x]       = data[offset+i+blockDim.x];

for(int stride=blockDim.x; stride>0 && i<stride; stride=stride/2)
{
  __syncthreads();
  partialSum[i] = partialSum[i] + partialSum[i+stride];
}

if(t==0)
  data[blockIdx.x] = partialSum[0] ;
```

→ approche à combiner avec le N-body simulation...

Shared Memory - Exemple 2 produit matriciel

- **Ex : produit matriciel**
  - opération de base pour résoudre Ax=b
  - C = A*B
  - parallélisme naturel :
    → 1 tache = calcul d'un élément C(i,j)
    = 1 produit scalaire
  - → accès mémoires prohibitifs

- **Exercice**
  - WIDTH=1024
  - #loads = ??
  - #(mul+add) = ??



Shared Memory - Exemple 2 produit matriciel

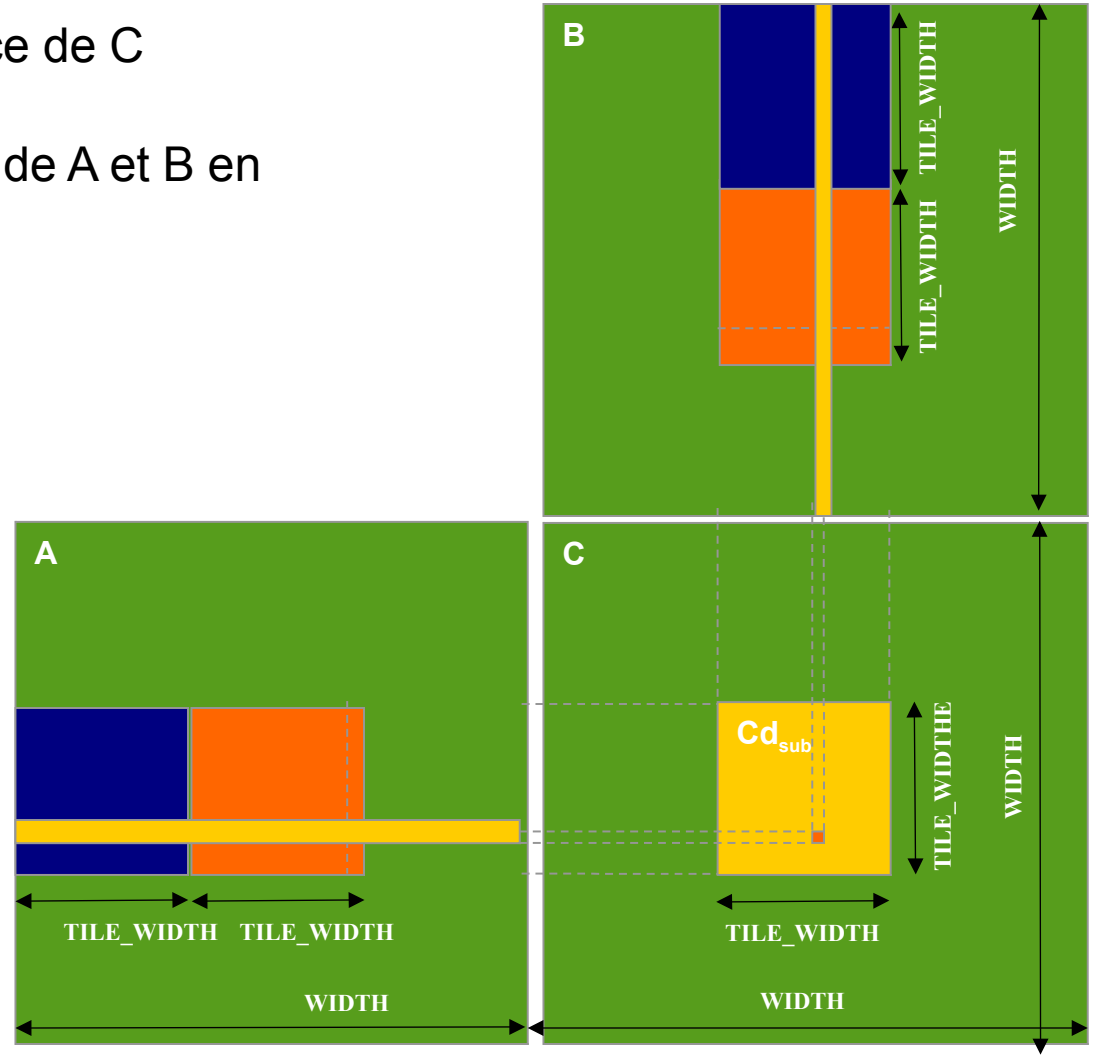
```
int i = threadIdx.x + blockIdx.x*blockDim.x,
    j = threadIdx.y + blockIdx.y*blockDim.y;

float c = 0 ;

for (int k = 0; k<N; k++)
{
  c += A[i+(j+k)*N] * B[i+k+j*N];
}

C[i+j*N] = c ;
```

◄— **slow**

Shared Memory - Exemple 2 produit matriciel

- **Idée**
  - considérer une sous matrice de C = 1 bloc de threads
  - pré-charger les sous blocs de A et B en shared memory

- **Exercice**
  - WIDTH=1024
  - TILE_WIDTH=32
  - #loads = ??
  - #madd = ??



Shared Memory - Exemple 2 produit matriciel

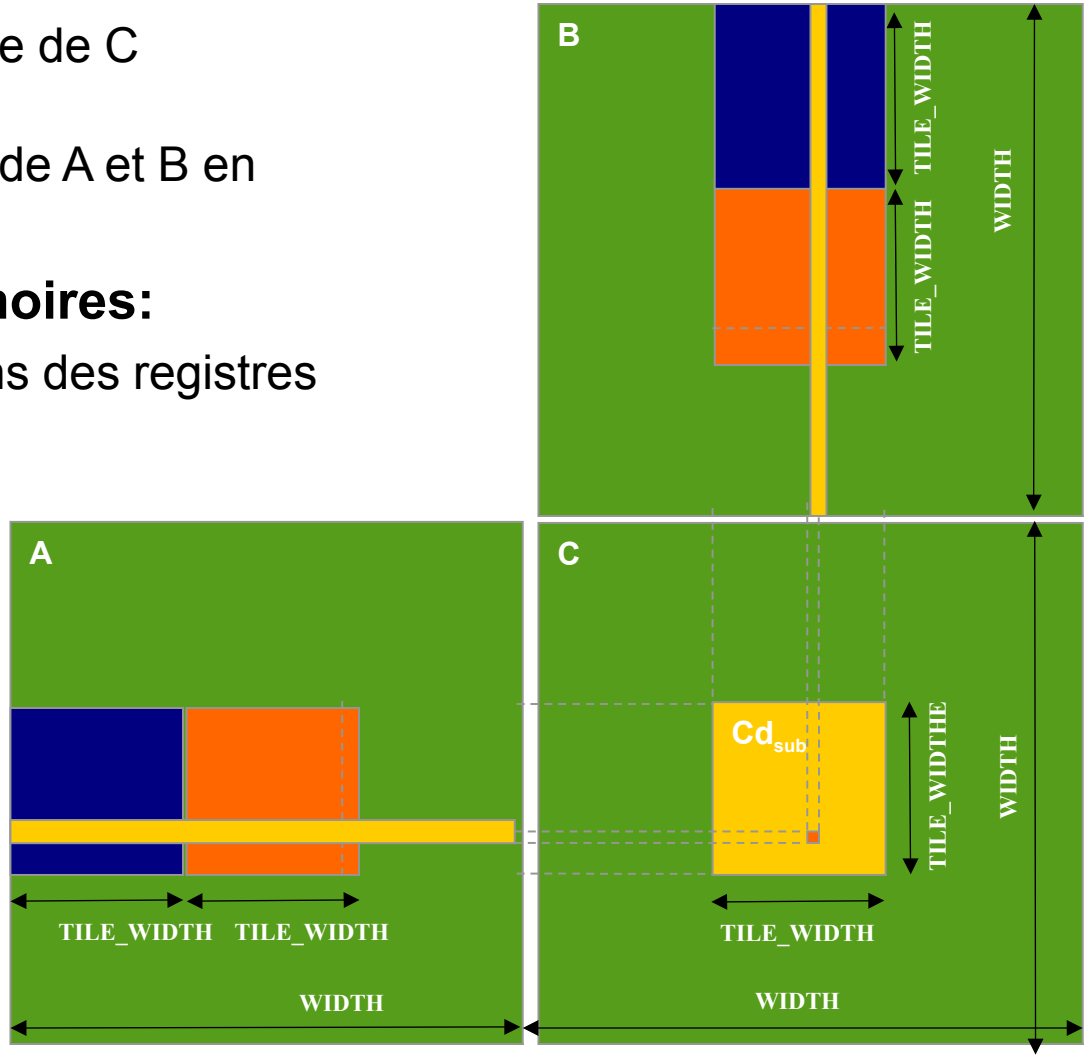
```
int i = threadIdx.x, j = threadIdx.y;
int oi = blockIdx.x*blockDim.x, oj = blockIdx.y*blockDim.y ;
__shared__ float tileA[T*T], tileB[T*T];

float c = 0 ;

for (int k = 0; k<N; k+=T)
{
   tileA[i+j*T] = A[oi+i+(k+j)*N] ;
   tileB[i+j*T] = B[k+i+(j+oj)*N] ;

   __syncthreads();

   for(int k1 = 0 ; k1<T; ++k1)
     c += tileA[i+k1*T] * tileB[k1+j*T] ;
}
```

← **slow**

← **fast**

Shared Memory - Exemple 2 produit matriciel

- **Idée**
  - considérer une sous matrice de C = 1 bloc de threads
  - pré-charger les sous blocs de A et B en shared memory
- **Masquage des accès mémoires:**
  - charger les blocs bleus dans des registres
  - pour chaque bloc
    - copier les blocs des registres vers la shared memory
    - copier les blocs suivant (oranges) dans des registres
    - calculer Cij pour les blocs courants



Shared Memory - Exemple 2 produit matriciel

```
int i = threadIdx.x, j = threadIdx.y;
int oi = blockIdx.x*blockDim.x, oj = blockIdx.y*blockDim.y ;
__shared__ float tileA[T*T], tileB[T*T];

float a = A[oi+i+j*N],
      b = B[i+(j+oj)*N];
      c = 0 ;

for (int k = 0; k<N; k+=T)
{
   tileA[i+j*T] = a ;
   tileB[i+j*T] = b ;

   __syncthreads();

   a = A[oi+i+(j+k)*N] ;
   b = B[i+k+(j+oj)*N] ;

   for(int k1 = 0 ; k1<T; ++k1)
     c += tileA[i+k1*T] * tileB[k1+j*T] ;
}
```

Parallel prefix-sum

Parallel prefix sum

- **Prefix sum**
  - entrée un tableau A[i], i=0..n-1
  - en sortie un tableau B, tq : B[i] = A[0] + … + A[i-1]
    - = réduction de chacun des préfixes (autres opérateurs : min, max, etc.)

input	2	4	3	1	2	7	1	4	2	5

prefix sum	0	2	6	9	10	12	19	20	24	26	31

  - algo séquentiel trivial
  - version parallèle : plus complexe !

    → implémentation disponible dans le SDK de Cuda (thrust) :)

Belloch, 1990, "Prefix Sums and Their Applications

Parallel prefix sum

- **algorithme fondamental**
- **domaines d'applications :**
  - algo de tri : radix sort, quicksort
  - comparaison de chaines de caractères, analyse lexical
  - **compacter, générer des données**
  - évaluation des polynômes (x, x*x, x*x*x, x*x*x*x, ...)
  - opérations sur les arbres
  - histogramme
  - MapReduce
  - etc.
  - → ***Belloch, 1990, "Prefix Sums and Their Applications***

Ex. : évaluer un polynôme

- **Passe 1 : évaluer tous les monômes 1, x, x^2, x^3, …**

input

x	x	x	x	x	x	x	x	x	x

prefix sum (with products)

1	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10

- **Passe 2 : multiplications par les coefficients**

monômes :

1	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10

* * * … *

coeffs :

2	4	-3	1	2	-1.2	1	3.4	2	5	-2

- **Passe 3 : réduction (somme)**

MapReduce avancé

- **Version séquentielle**

```
for(int i=0 ; i<N ; ++i)
  if(P(i))
    result = reduce(result, func(data[i])) ;
```

- **Version parallèle**
  - **Map**
    - appliquer la même fonction *func* aux données pour lesquels le prédicat P(i) est vrai
      - Parallélisme : 1 thread ↔ 1 élément avec P(i)==true
      - Si sous-ensemble non structuré
        → compacter les données via *prefix-sum*
  - **Reduce**
    - réduire les résultats du Map à une seule valeur

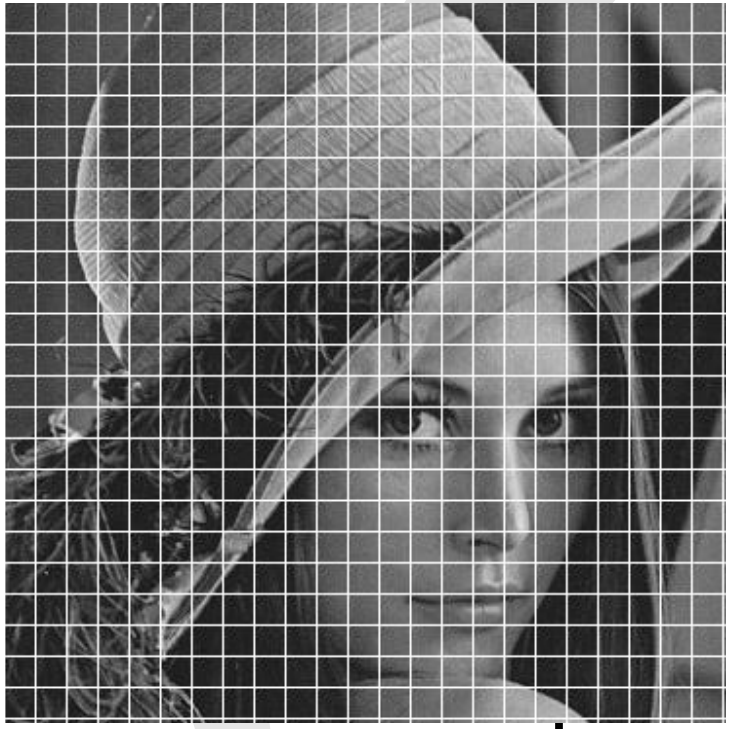
Branchement dynamique & gestion des données creuses

- **Exemple, on veux appliquer une fonction couteuse uniquement sur quelques éléments d'un tableau répartis de manière aléatoire**
  - ex : application d'un filtre sur les discontinuités d'une image, culling, raffinement, etc.
- **Kernel :**

```
__global__ void my_kernel(...) {
  int id = blockIdx.x*blockDim.x + threadIdx.x ;
  if( is_active(id) )
    compute(id) ;
}
```

  → inefficace à cause du SPMD
- **Solution en deux passes:**
  - « compacter » les données « actives »
  - nouveau kernel sans « if »
    - if(is_active(id)) → if (id<nb_actives)
- **Comment ? → « parallel prefix sum »**

Exemple : filtre median



input

output

sélection des
pixels à traiter
(sélection → préfix-sum → packing)

filtre
médian

Autres exemples

traitements spéciaux sur les silhouettes (ex., Sobel)

Prefix sum pour compacter des données creuses

- **Principe**

  1 – générer un tableau A[i] contenant des 1 et 0

  - A[i]==1 ↔ donnée #i est active

```
__global__ void selection(int* A) {
  int id = blockIdx.x*blockDim.x + threadIdx.x ;
  A[id] = is_active(id) ? 1 : 0 ;
}
```

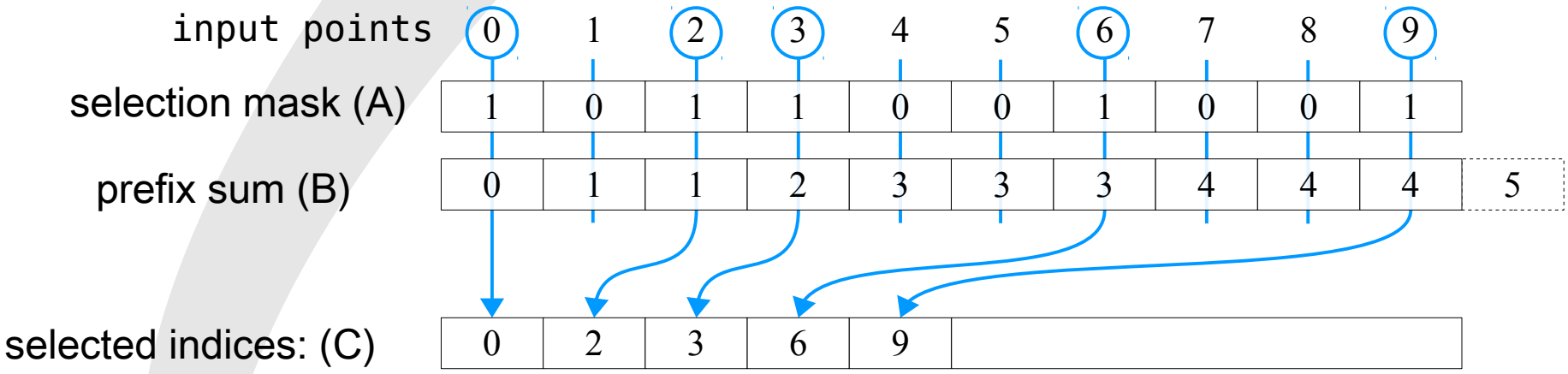
  2 – appliquer un préfix sum → B[i]

  - B[i] = nombre de données actives précédente à la donnée *i*
    = emplacement de la donnée *i* dans un tableau compact
  - B[N] = nombre de données actives

  3 – créer un tableau C compact des indices actifs *i* :

```
__global__ void pack_indices(const int* A, const int* B, int* C) {
  int id = blockIdx.x*blockDim.x + threadIdx.x ;
  if( A[i] ) C[ B[id] ] = id
}
```

Prefix sum pour compacter des données creuses



4 – appliquer notre calcul en utilisant C[]

```
__global__ void my_kernel(const int* C, ...) {
  int id = blockIdx.x*blockDim.x + threadIdx.x ;
  compute( C[id] ) ;
}
```

- **variantes :**
  - générer un tableau compact des indices i
  - et/ou compacter directement les données

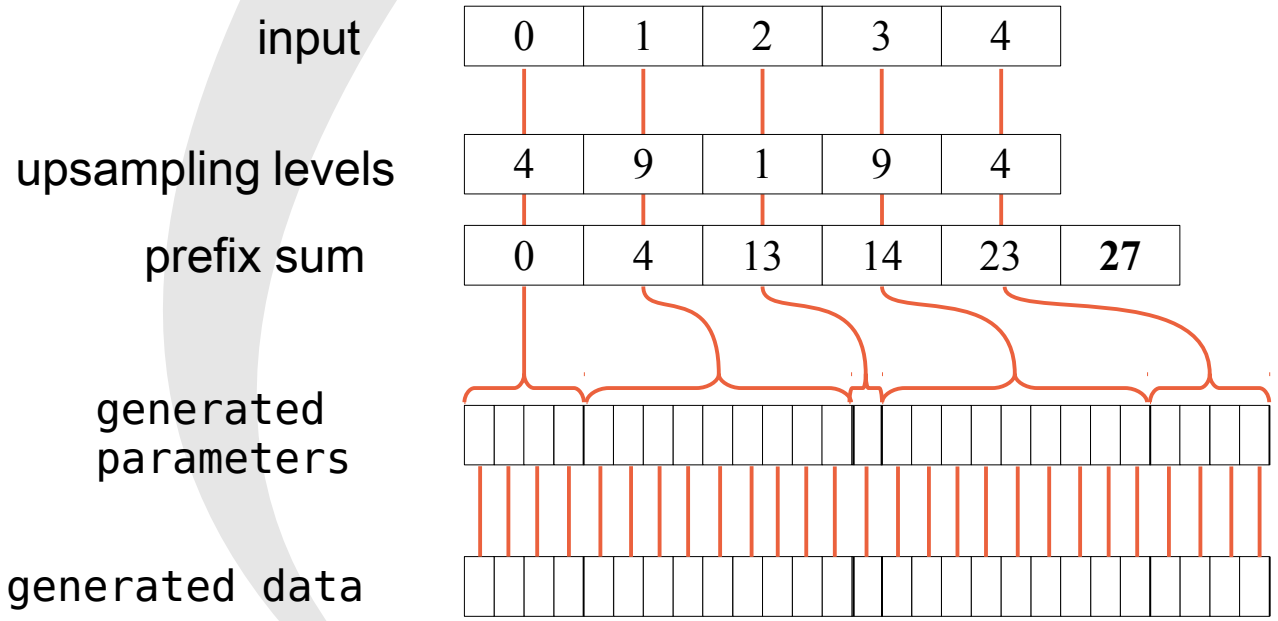
Prefix sum pour générer des données

- **Objectif**
  - chaque donnée initiale *i* génère un nombre variable M[i] de nouvelle donnée
  - ex : raffinement de maillage, subdivision, construction d'arbre, etc.
  - un thread par donnée initiale
    - deux problèmes :
      - où écrire les données ???
      - performances dues au SPMD

```
__global__ void generate_data(float* C, ...) {
  int id = blockIdx.x*blockDim.x + threadIdx.x ;
  int n = how_many(id) ;
  for(int i=0 ; i<n ; ++i)
  {
    c[ ??? ] = generate(id, i) ;
  }
}
```

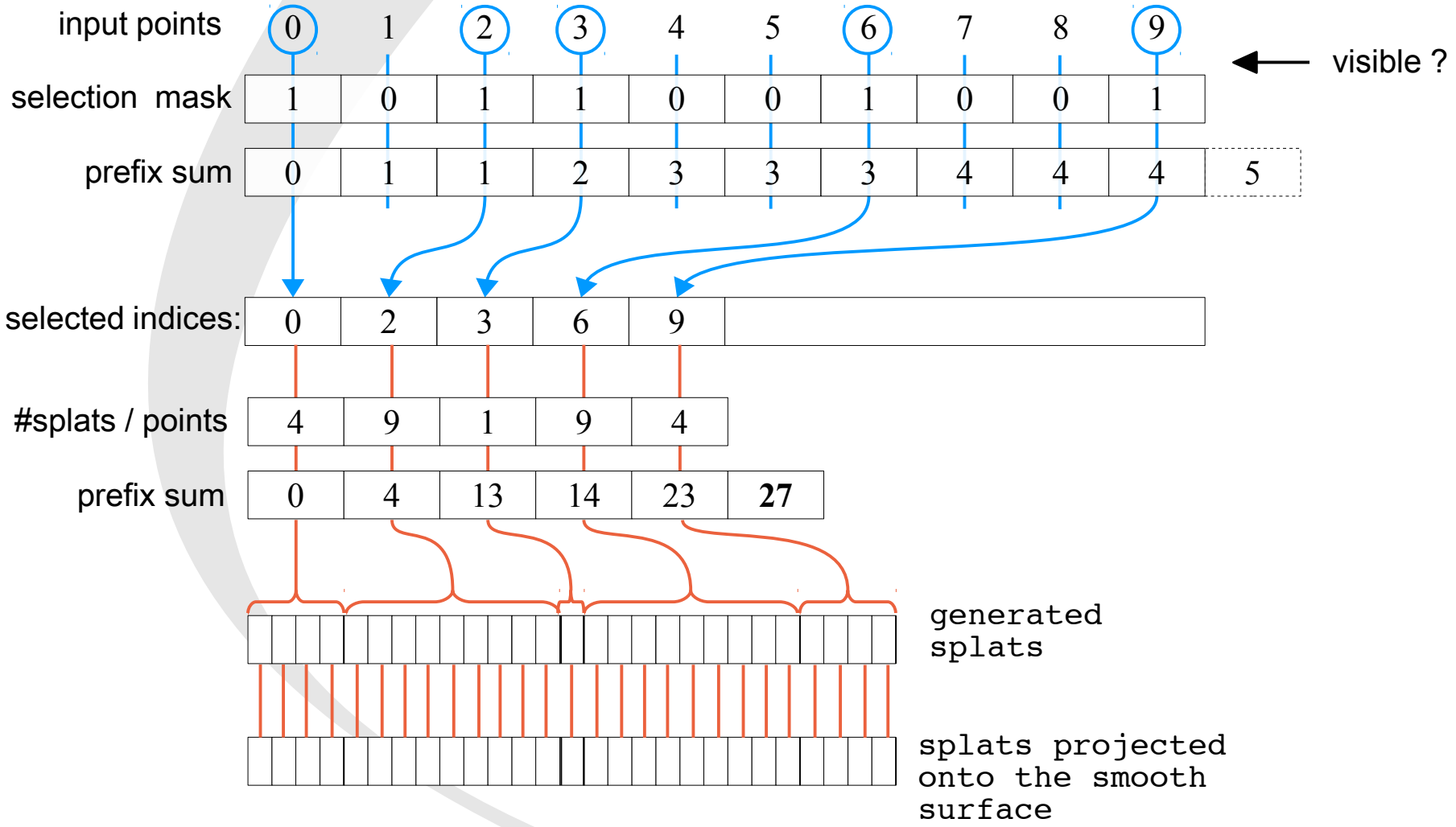
Prefix sum pour générer des données

- **Solution :**
  - où → prefix sum
  - SPMD → découplage génération des paramètres/évaluation

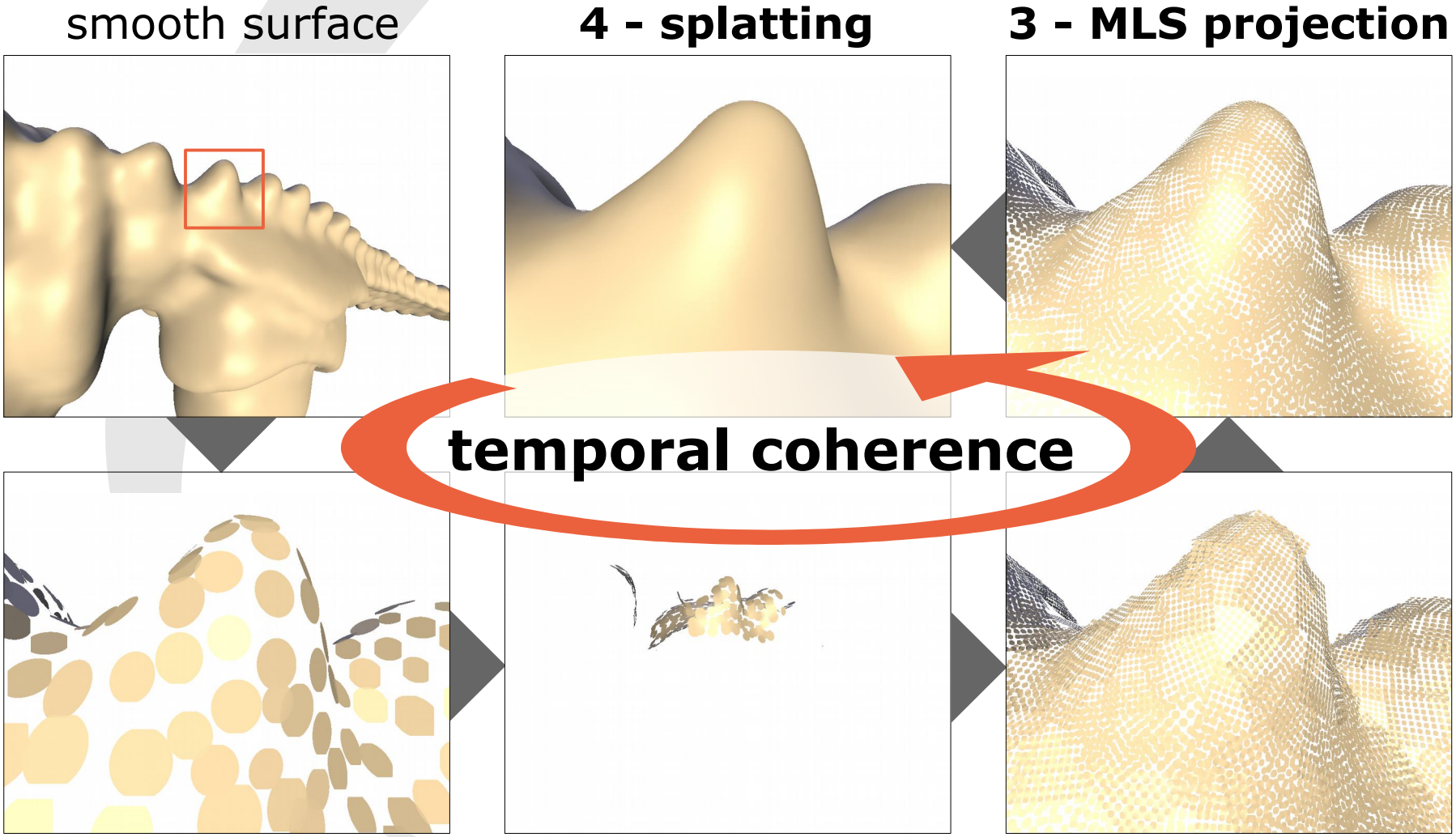


Exemple

- **Raffinement dynamique d'un nuage de points**

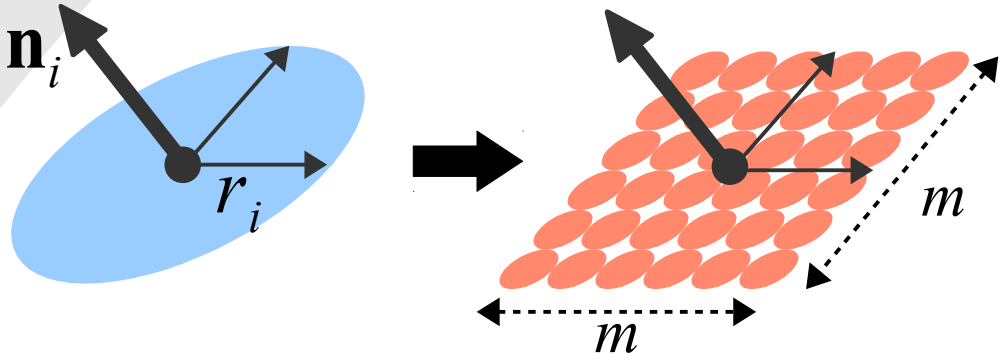


Raffinement dynamique d'un nuage de points

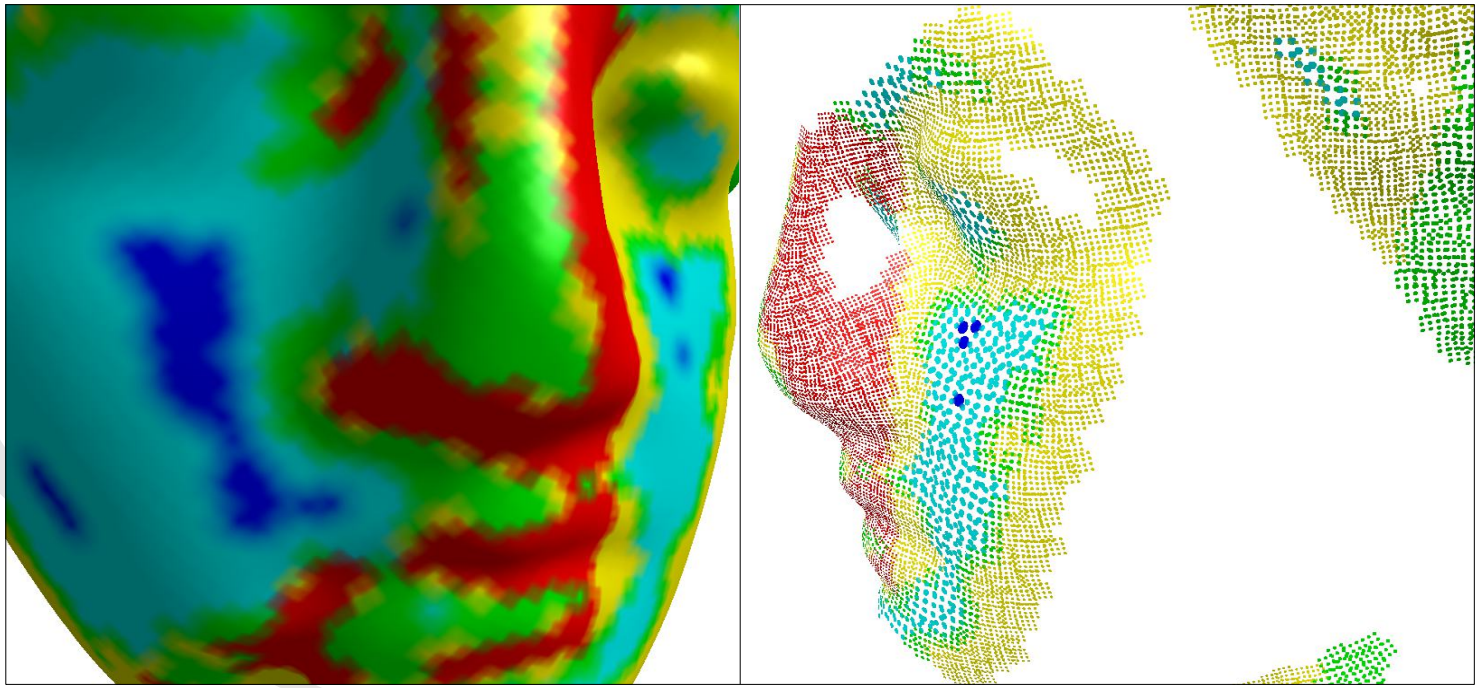


Raffinement dynamique d'un nuage de points

- **Upsampling :**

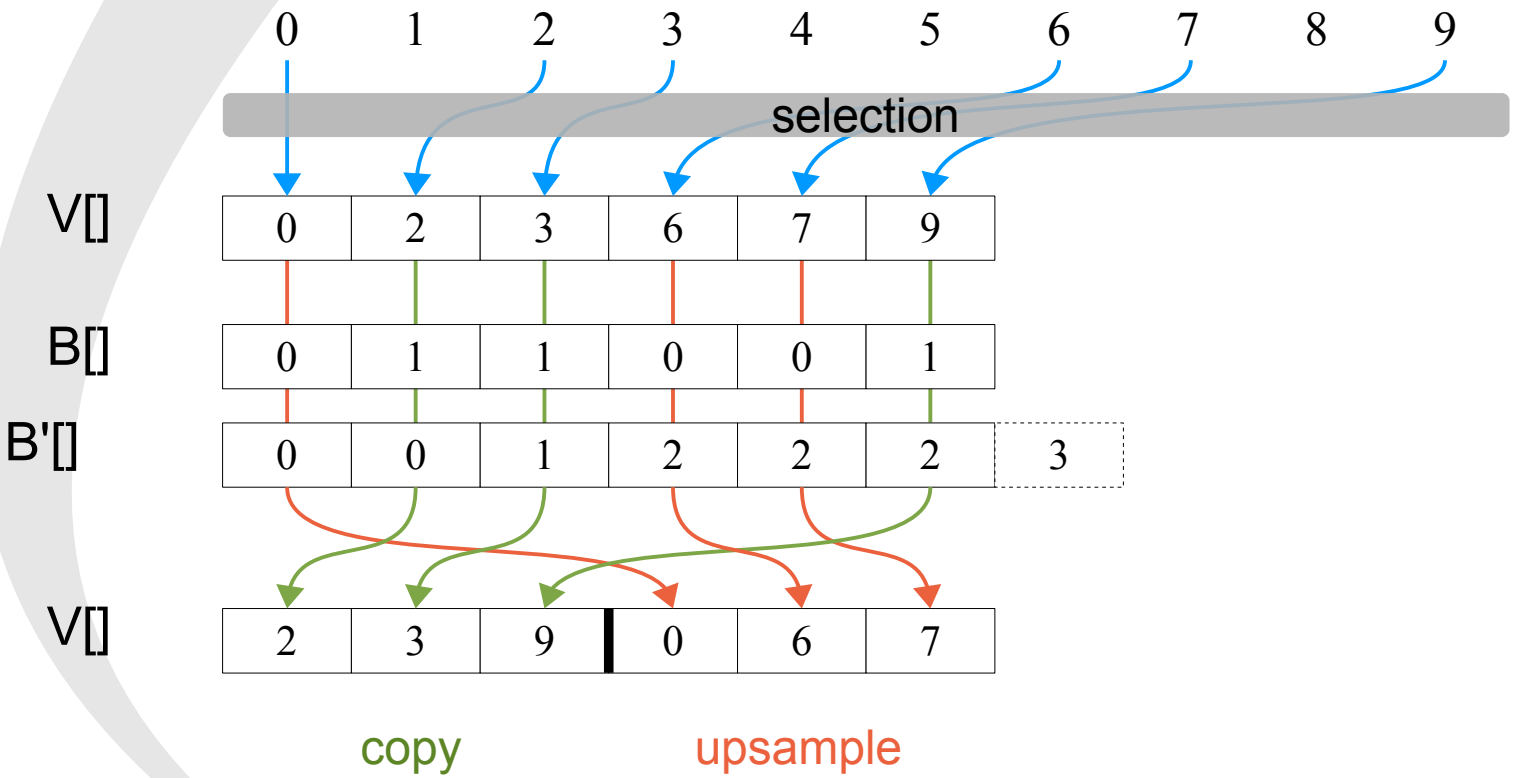


- **LOD :**



Raffinement dynamique d'un nuage de points

- **Cohérence temporelle**



Exercice : parallel prefix sum

reductions partielles

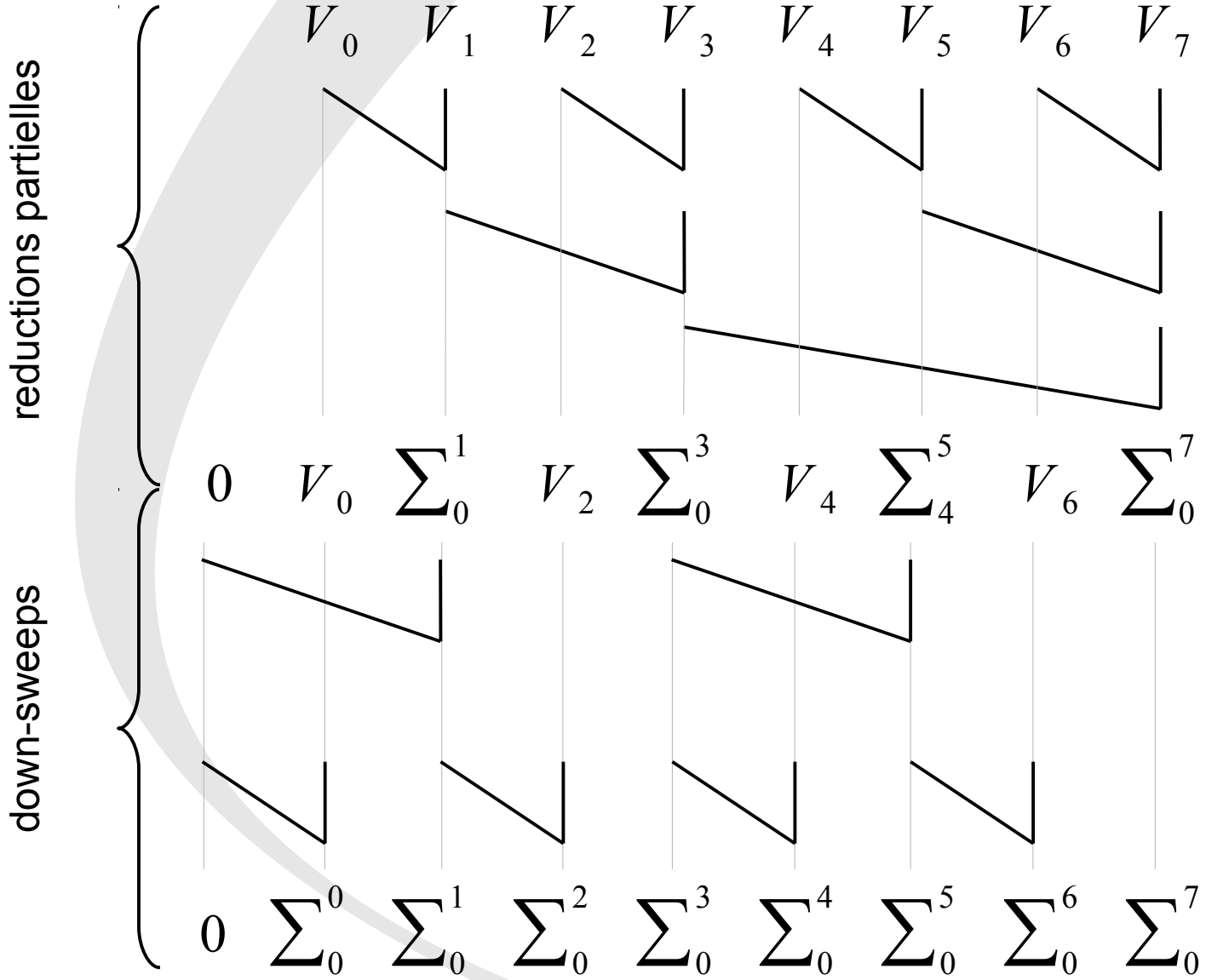
down-sweeps

$$V_0 \quad V_1 \quad V_2 \quad V_3 \quad V_4 \quad V_5 \quad V_6 \quad V_7$$

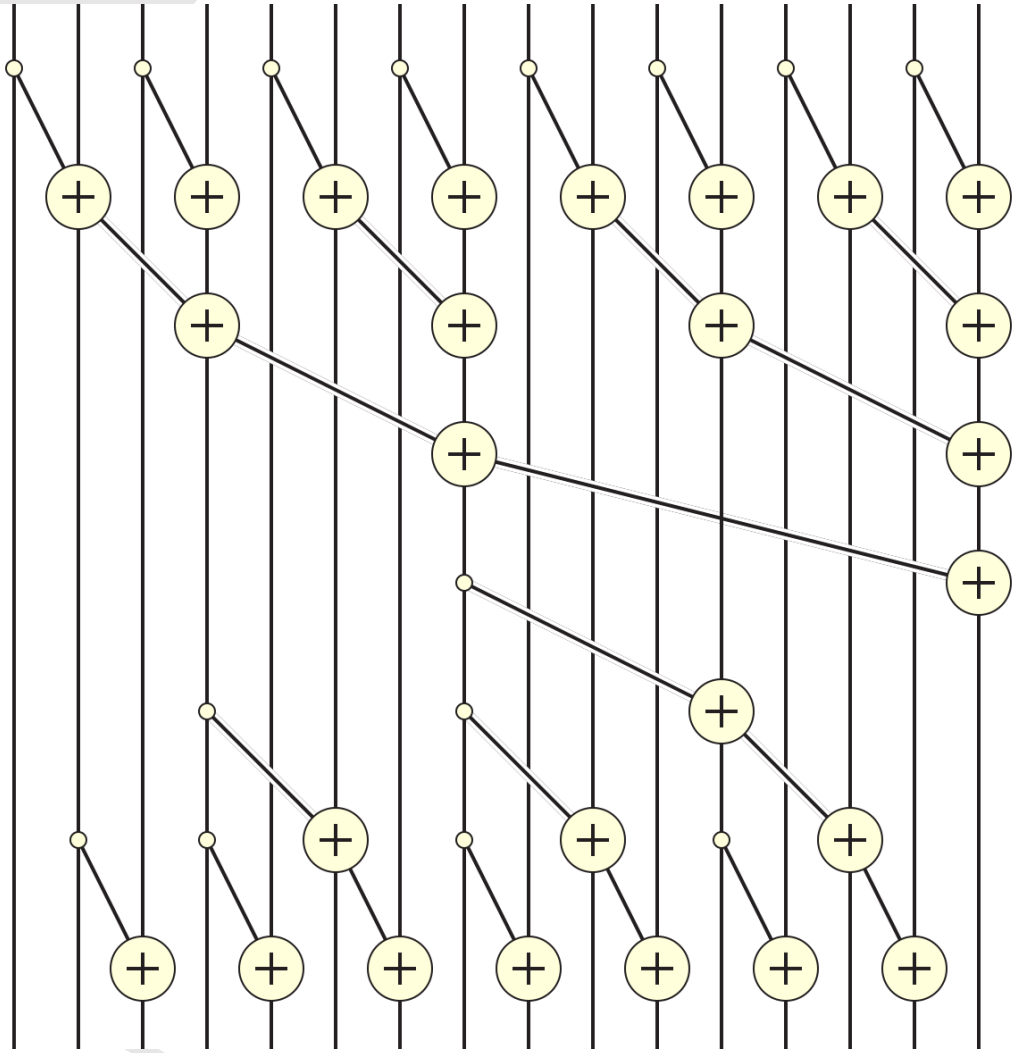
$$0 \quad V_0 \quad \sum_0^1 \quad V_2 \quad \sum_0^3 \quad V_4 \quad \sum_4^5 \quad V_6 \quad \sum_0^7$$

$$0 \quad \sum_0^0 \quad \sum_0^1 \quad \sum_0^2 \quad \sum_0^3 \quad \sum_0^4 \quad \sum_0^5 \quad \sum_0^6 \quad \sum_0^7$$

Exercice : parallel prefix sum

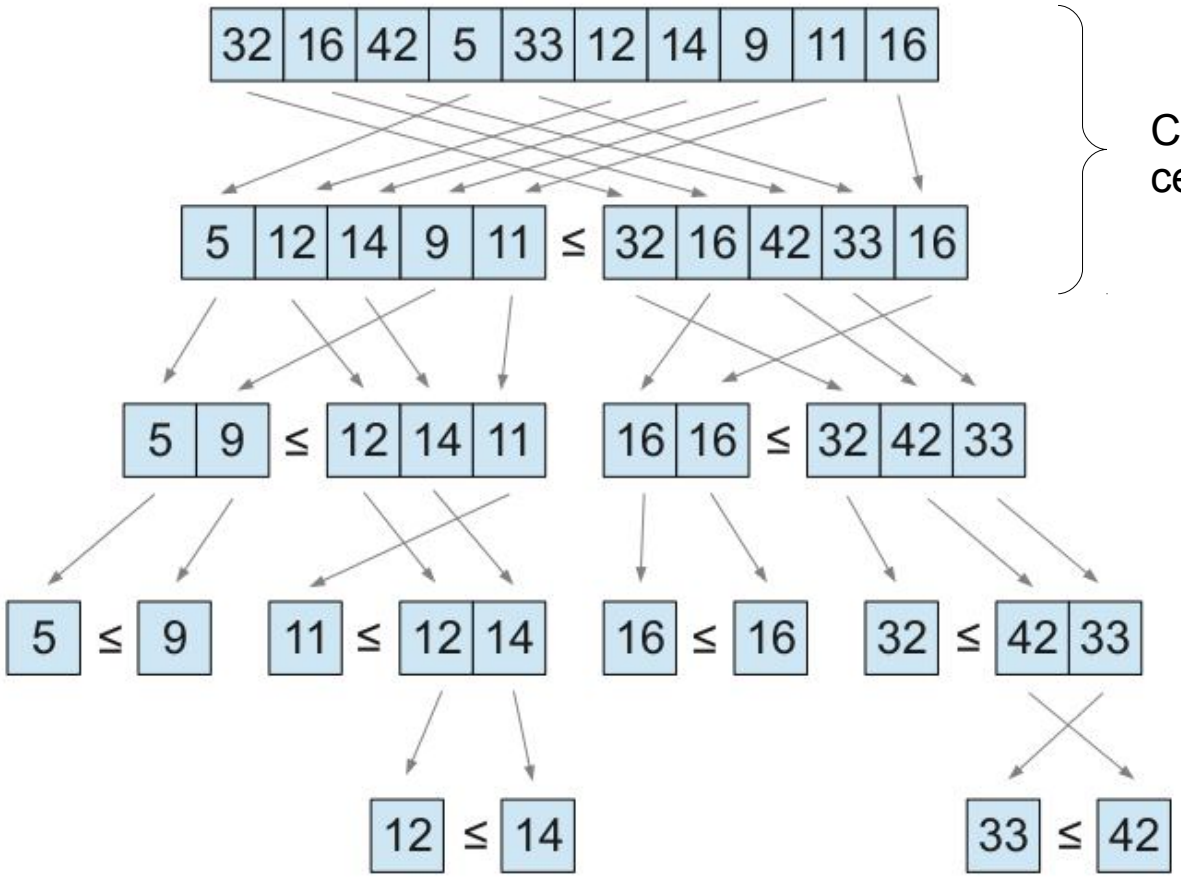


Exercice : parallel prefix sum



Exercice : Quick Sort

- **Rappels :**



Comment paralléliser ce tri partiel ?

Autres interfaces de programmations

- **CUDA : C/C++, GPU Nvidia**
  - Interfaces : Fortran, Python, Java, MatLab, etc.
  - Bibliothèques : thrust, CuBlas, CuFFT, etc.
- **OpenCL : C,**
- **OpenACC : C/Fortran, Générique**
  - Compilateur PGI, (implémentation dans gcc en cours)
  - OpenMP 4

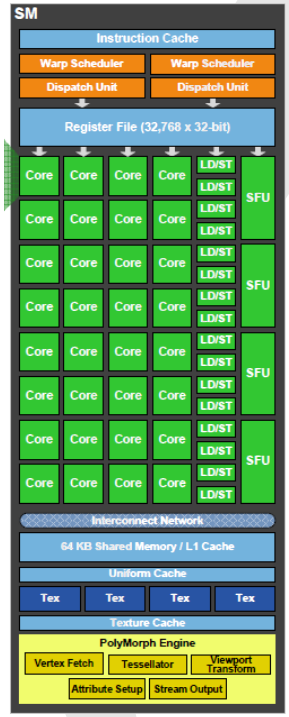
```
Vec3 a[N] ;
Vec3 b ;
float c[N] ;

#pragma acc parallel for shared(N,a,b,c)
for (i=0; i<N; i++)
  c[i] = a[i].x*b.x + a[i].y*b.y + a[i].z*b.z;
```

```
#include <stdio.h>
#define N 1000000

int main() {
  double pi = 0.0f; long i;
  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N) ;
    pi += 4.0/(1.0+t*t);
  }
  printf("pi=%16.15f\n",pi/N);
  return 0;
}
```

Autres architectures *many-cores*

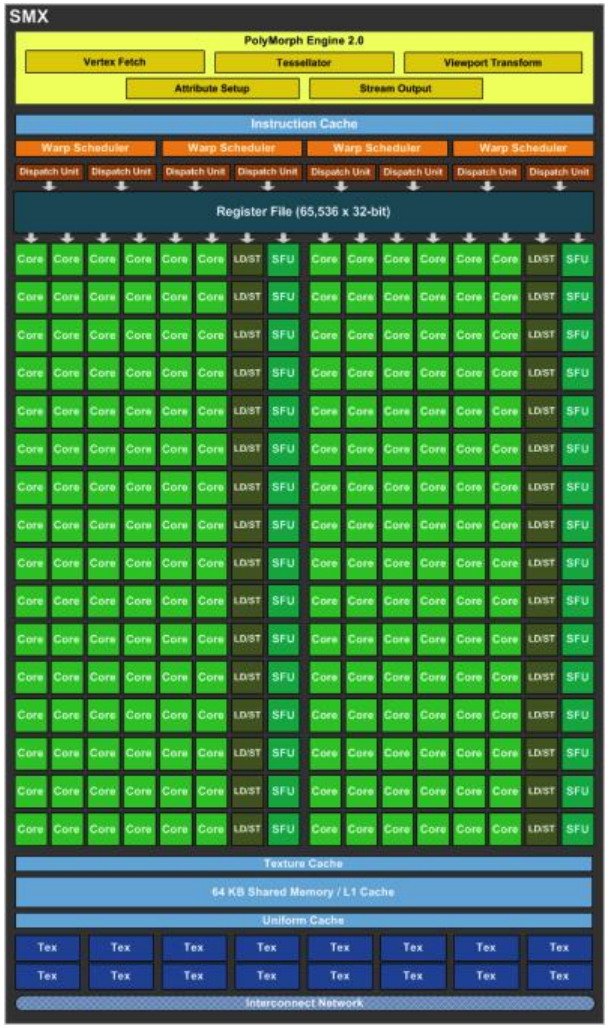
Nvidia GTX 680

GTX 580



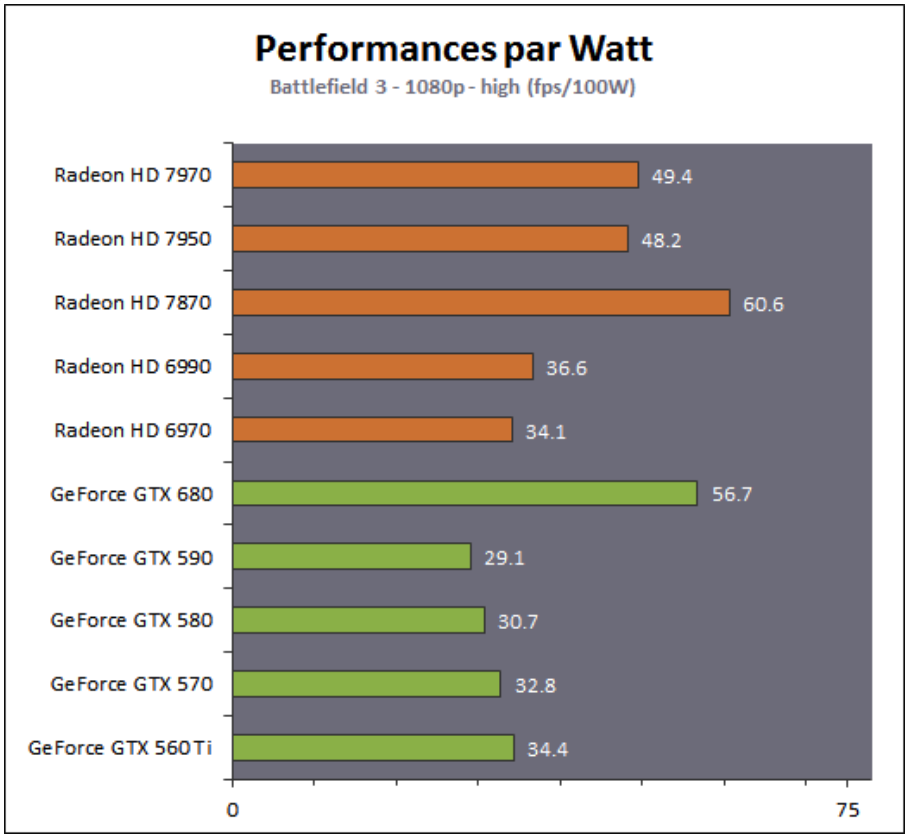
- SM : 32 cores
- fréquence double
  - 64 ops/cycle
  - 2 warps actifs
- gravure : 40nm
- 512 cores au total

GTX 680



- gravure : 28nm
- 192 cores !
  - 192 ops/cycle
  - 4 warps actifs
- « super scalaire »
- ordonnancement statique des opérations déterministes
- optimisé pour le graphique
- ~1500 cores au total

Nvidia GTX 680



Radeon 7970

- **Radeon**
  - warp size : 64
- **Evolutions**
  - 5800 : 5 instructions de front sur 16 éléments (threads, pixels, vertices, etc.)
  - 6900 : 4 instructions de front sur 16 éléments (threads, pixels, vertices, etc.)
    - 1 unité SIMD de 64 cores
    - 8 cycles/instruction
      → 2 warps actifs (128 éléments)
      → 8 cycles : 128 éléments traités avec 4 opérations en parallèle
  - 7900 : 1 instruction sur 16 éléments (threads, pixels, vertices, etc.)
    - 4 unités SIMD de 16 cores
    - 4 cycles/instruction
      → 4 warps actifs (256 élements)
      → 4 cycles : 256 éléments traités avec 1 seule opération
    - ~2000 cores au total

Radeon 7970

1 instruction scalaire à exécuter :
CU VLIW 4 : 16 / 24 / 40 cycles
CU GCN : **11** / **11** / **15** cycles

100 instructions scalaires à exécuter :
CU VLIW 4 : 408 / 808 / 1608 cycles
CU GCN : **207** / **207** / **407** cycles

1 instruction vec3 à exécuter :
CU VLIW 4 : **16** / 24 / 40 cycles
CU GCN : 19 / **19** / **31** cycles

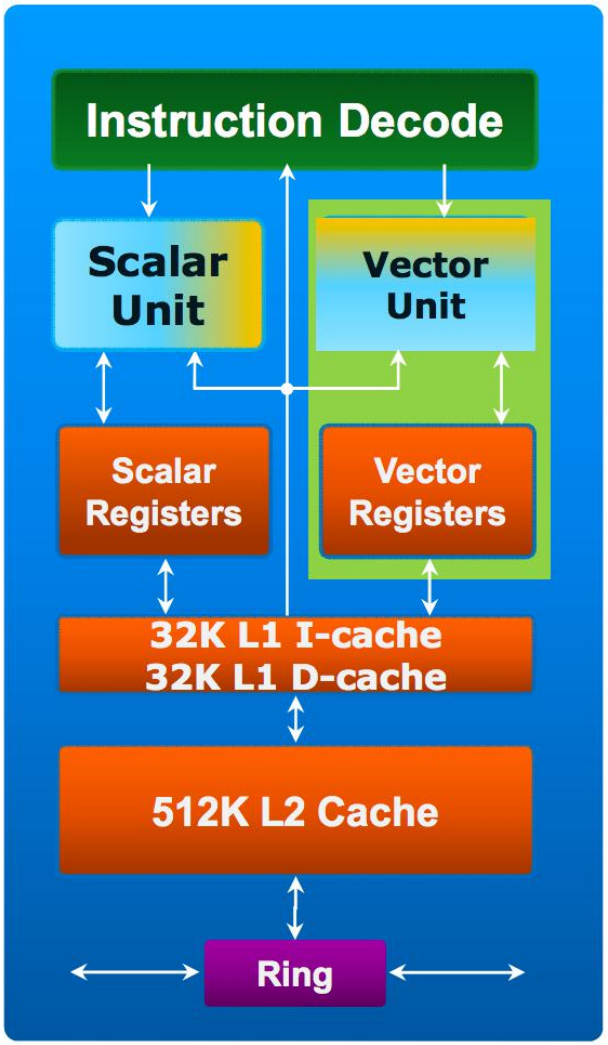
100 instructions vec3 à exécuter :
CU VLIW 4 : **408** / 808 / 1608 cycles
CU GCN : 607 / **607** / **1207** cycles

1 instruction vec4 à exécuter :
CU VLIW 4 : **16** / 24 / 40 cycles
CU GCN : 23 / 23 / 39 cycles

100 instructions vec4 à exécuter :
CU VLIW 4 : **408** / 808 / 1608 cycles
CU GCN : 807 / 807 / 1607 cycles

- VLIW = Radeon 6900
- GCN = Radeon 7900

Intel Xeon Phi



>50 in-order cores

- Ring interconnect

64-bit addressing

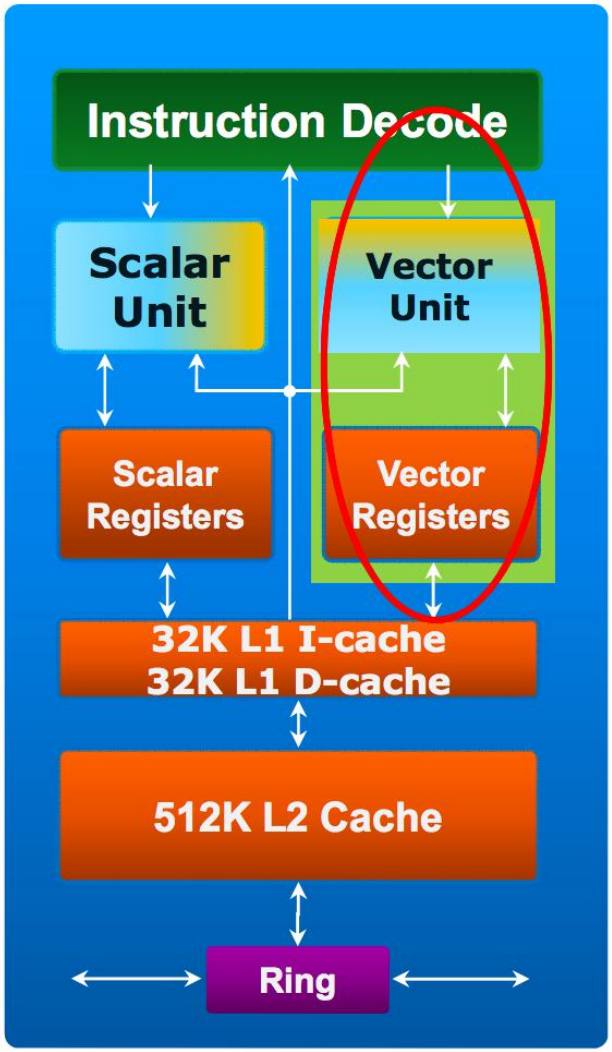
Scalar unit based on Intel® Pentium® processor family

- Two pipelines
  - Dual issue with scalar instructions
- One-per-clock scalar pipeline throughput
  - 4 clock latency from issue to resolution

4 hardware threads per core

- Each thread issues instructions in turn
- Round-robin execution hides scalar unit latency

Intel Xeon Phi



Optimized

- Single and Double precision

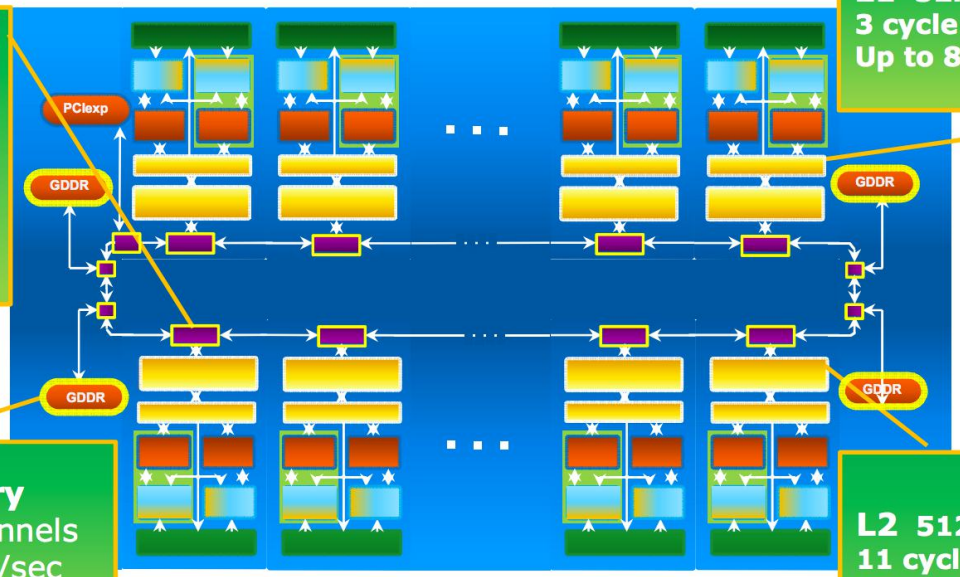
All new vector unit

- 512-bit SIMD Instructions – not Intel® SSE, MMX™, or Intel® AVX
- 32 512-bit wide vector registers
  - Hold 16 singles or 8 doubles per register

Fully-coherent L1 and L2 caches

Intel Xeon Phi

Individual cores are tied together via fully coherent caches into a bidirectional ring



Bidirectional ring
115 GB/sec
Distributed Tag Directory (DTD) reduces ring snoop traffic
PCIe port has its own ring stop

L1 32K I- D-cache per core
3 cycle access
Up to 8 concurrent accesses

GDDR5 Memory
16 memory channels
- Up to 5.5 Gb/sec
8 GB 300ns access

L2 512K cache per core
11 cycle best access
Up to 32 concurrent accesses

Takeaway: Parallelization and data placement are important