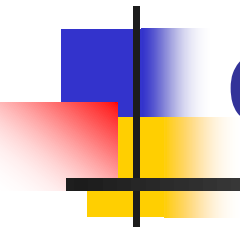
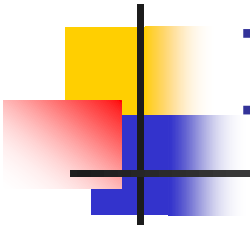


Architecture et programmation d'accélérateurs matériels

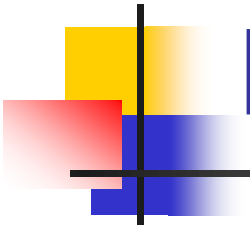
Cours 1 : Architecture et modèles de programmation

Julien Jaeger
Patrick Carribault
julien.jaeger@cea.fr



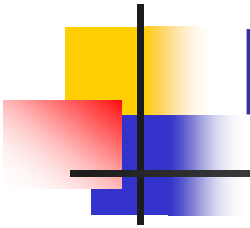
Introduction

- Thématique : *accélérateurs*
  - Etat de l'art
  - Architecture des accélérateurs
  - Programmation des accélérateurs
  - Optimisations caractéristiques
- Pré-requis
  - Notion de base architecture
  - Programmation impérative
  - Notion de programmation parallèle



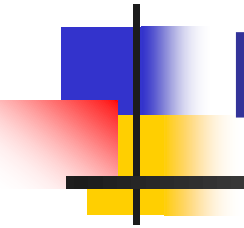
Plan général

- But :
  - Introduction aux accélérateurs
  - Expertise en programmation CUDA
- Plan :
  - Architecture
  - Modèle de programmation
  - CUDA et OpenCL
- Intervenant
  - Patrick Carribault (*patrick.carribault@cea.fr*)
  - Julien Jaeger (*julien.jaeger@cea.fr*)
  - Hugo Taboada

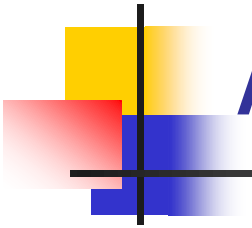


Plan du cours

- Panorama des accélérateurs
- Architecture de GPGPU
  - NVIDIA Kepler
- Modèles de programmation pour accélérateurs
- NVIDIA CUDA (partie 1)

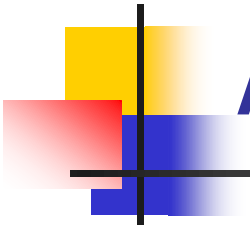


Panorama des accélérateurs



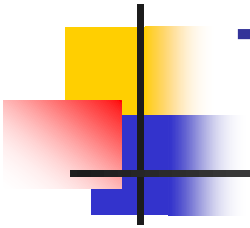
Accélérateurs

- Définition
  - Ressource distincte (située sur la même puce que le processeur principal ou non) définie pour un ensemble différents d'objectifs que le processeur central
- En clair :
  - Processeur dédié à une classe spéciale d'applications
  - Permet d'obtenir de meilleures performances dans son domaine de prédilection



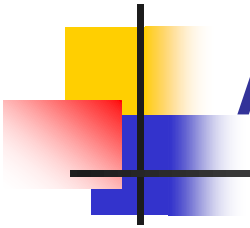
Accélérateurs – Interconnexion

- Les accélérateurs sont reliés au(x) processeur(s) principal(aux) grâce à une *interconnexion*
- Comportement similaire à un périphérique
  - Le processeur central contrôle l'accès
  - Il exécute également la partie principale du code
- Plusieurs types d'interconnexion
  - Accélérateurs déportés
  - Accélérateurs sur la puce principale



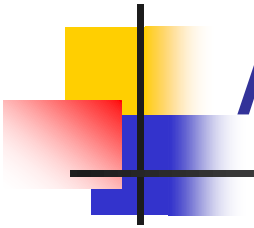
Types de connexions

- Via le bus système
  - GPGPU nVidia, MIC Intel
  - Avantages
    - Faible coût
    - Intégration aisée
  - Inconvénients
    - Performances modestes dûes à la généricité du bus
    - Bande passante et latence sous-optimale
- Via le bus du processeur
  - IBM Cell , Architecture Fusion d'AMD/ATI
  - Avantages
    - Accès rapide
    - Possibilité de partager des ressources avec le CPU
  - Inconvénients
    - Intégration complexe
    - Dépendance avec le processeur



Architecture des accélérateurs

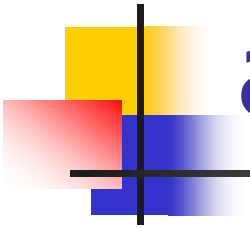
- Microarchitecture
  - Dirigée par la diversité des applications à exécuter ainsi que du domaine de prédilection
  - Simplification par rapport aux CPUs
    - Exécution dans le désordre, prédicteur de branchements, …
- Parallélisme exploité
  - Généralement à grain très fin
  - La plupart exploite une forme de *Single Instruction Multiple Data*
- Mais
  - Une partie *importante* du travail est laissée à la pile logicielle et au développeur de codes



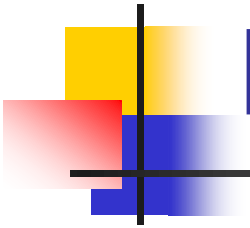
Architecture des accélérateurs

- Hiérarchie mémoire
  - Mémoire controllée par le logiciel/programmeur *(scratchpad memory)*
  - Cache géré par le matériel (comme sur un CPU généraliste)
  - Interconnexion matérielle entre les différents processeurs de calcul
  - Bande passante importante entre l'accélérateur et la mémoire globale
- Résumé
  - Possibilités intéressantes pour le calcul haute performance
  - Mais beaucoup d'aspects à gérer *à la main*.
  - Ceci influence directement la façon de programmer les accélérateurs

Programmation des accélérateurs

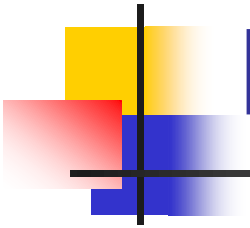


- Vision d'un accélérateur comme un périphérique *esclave*.
- Le processeur principal contrôle l'accès aux ressources de calculs spécialisées
  - Processeur principal (CPU) : *hôte* ou *host*
  - Accélérateur : *device*
- Concepts généraux
  - Transferts des données
  - Exécution d'un code déporté dans un langage différent
  - Gestion manuel de l'exécution
  - Interrogation de l'accélérateur sur son état
  - Rapatriement des données sur l'hôte



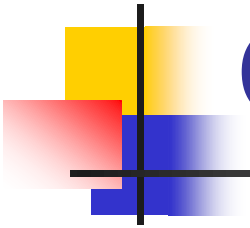
Panorama des accélérateurs

- Processeurs graphiques
  - GPGPUs (General Purpose Graphics Processing Units)
  - NVIDIA
  - AMD/ATI
- Accélérateurs dédiés
  - Cell (IBM)
  - MIC ou Xeon Phi (Intel)
- Architectures reconfigurables
  - FPGA



Processeurs graphiques

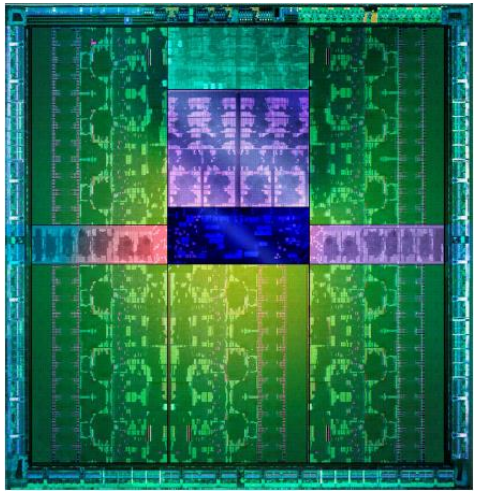
- Cartes graphiques initialement pour l'affichage 2D et 3D sur écran
  - Processeur(s) graphique(s) dédié(s)
- Capacités de calcul impressionnantes
  - Optimisation pour le rendu d'image (objectifs différents des processeurs généralistes)
  - Idéal pour le traitement sur des données régulières
- Utilisation pour le calcul scientifique
  - GPGPU (Generale Purpose Graphics Processing Unit)

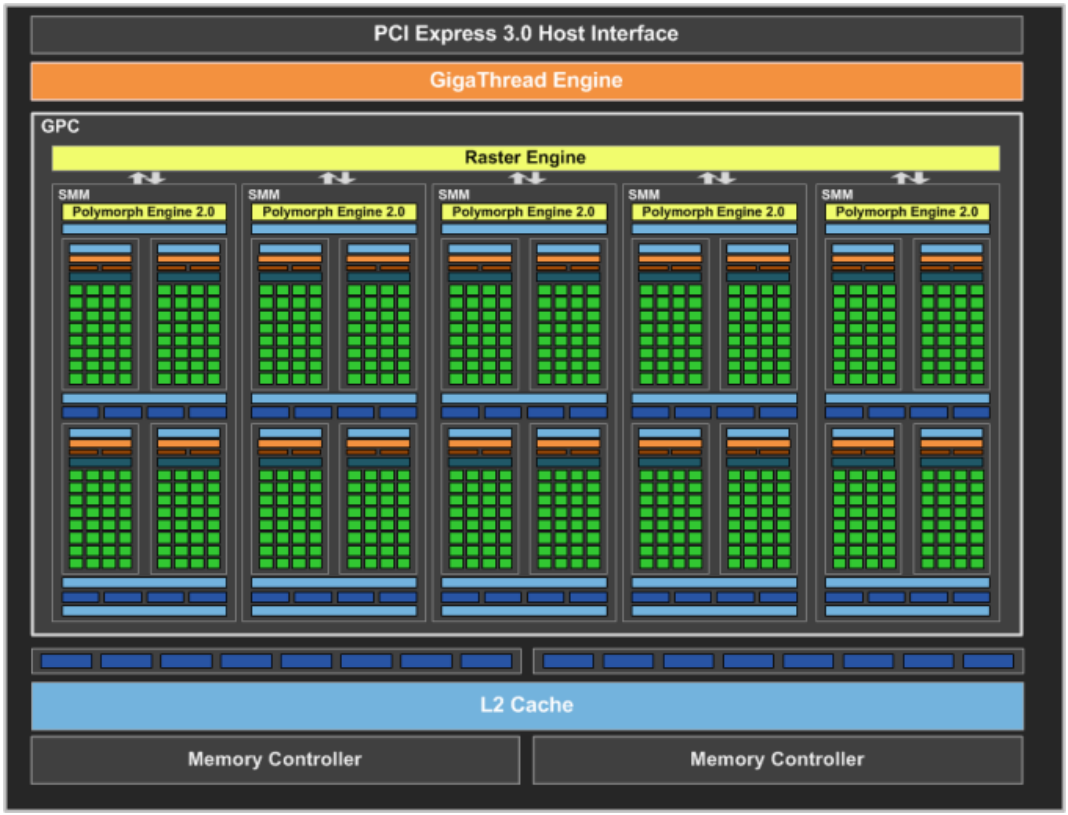


GPGPU

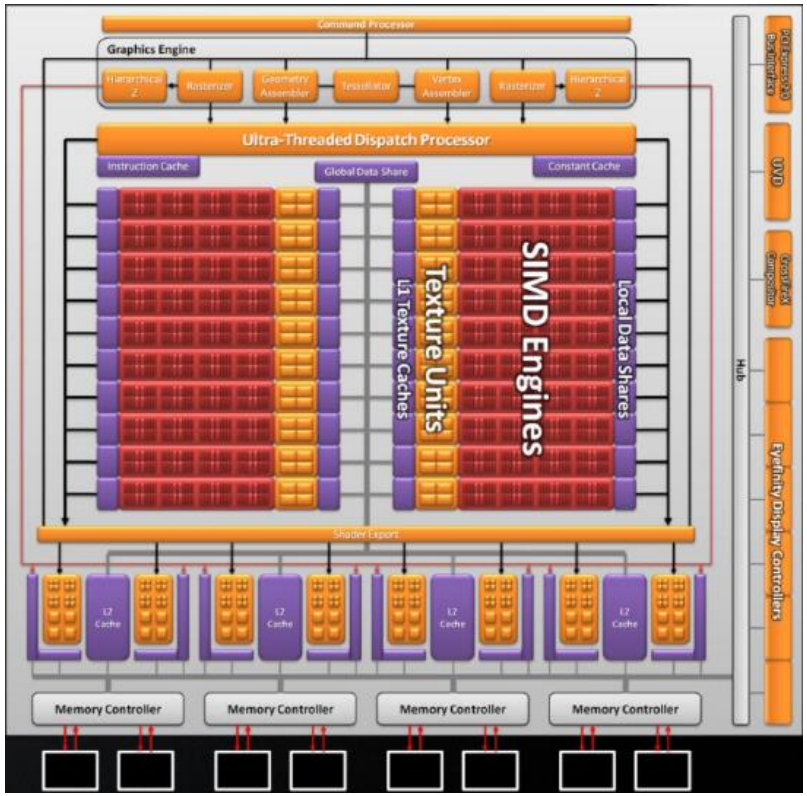
- Cartes graphiques sur un bus système
  - Interconnexion sur la carte mère
  - Communications via le port PCI-express
- Possibilité de connecter plusieurs cartes graphiques
  - Programmation Multi-GPU
  - Utilisé actuellement dans les supercalculateurs hybrides

Architecture Maxwell NVIDIA



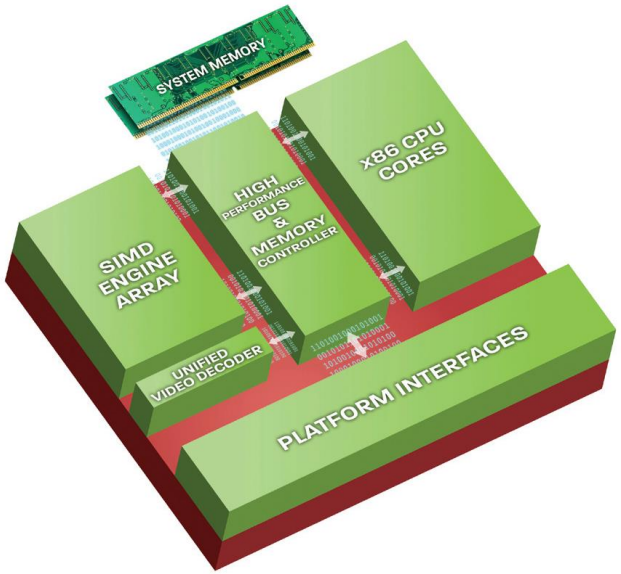


Architecture ATI Cypress

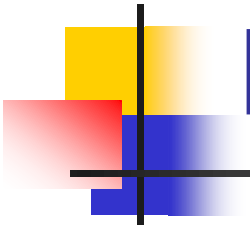


- Ensemble d'unités vectorielles
  - SIMD Engines
- Hiérarchie mémoire
  - Cache L2
- Performances crêtes
  - 2,72 TFlops SP
  - 0.54 TFlops DP

Architecture Fusion (Llano)



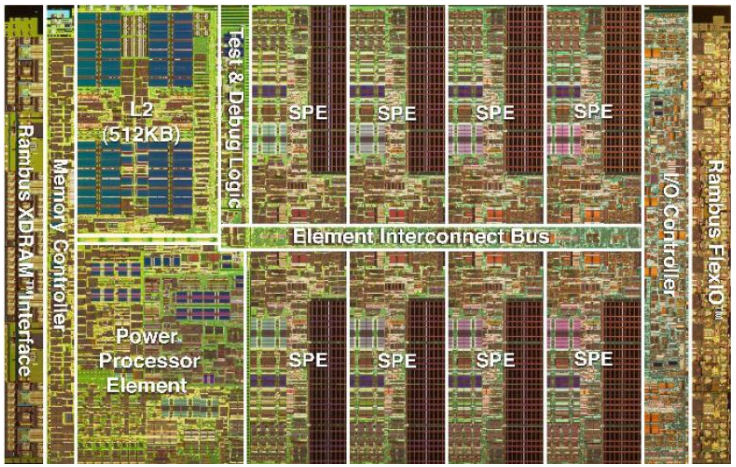
- Nom de code de la prochaine architecture d'AMD/ATI : Fusion
- Design d'un APU (Accelerator Processing Unit)
- Sur la même puce :
  - CPU
  - Accélérateur de type GPU
- Première génération disponible
  - Llano avec technologie Bulldozer ou Bobcat
- `http://fusion.amd.com`



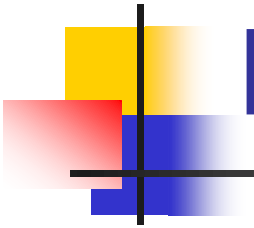
MIC (Intel)

- Accélérateur proposé par Intel
  - Many-Integrated Core (MIC)
- Introduction en 2008
  - Conférence SigGraph 2008
  - Connu sous le nom *Larabee*
- Architecture
  - Coeurs x86
  - Support de l'*hyperthreading*
  - Cohérence de cache
- Version précédente : KNC
- Version courante : KNL

Le processeur CELL (IBM)

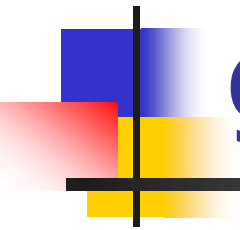


- Développé conjointement pas IBM, Sony et Toshiba
- 8 cœurs 64-bit, à calculs flottants
  - Appelés *Synergistic Processor Elements* (SPEs).
- Un processeur maître PowerPC 64-bit capable d'exécuter deux threads
- SPEs peuvent traiter des opérandes 128-bit operands, décomposées en quatre mots 32-bits (SIMD).
- Le banc de registres peut stocker 128 opérandes.



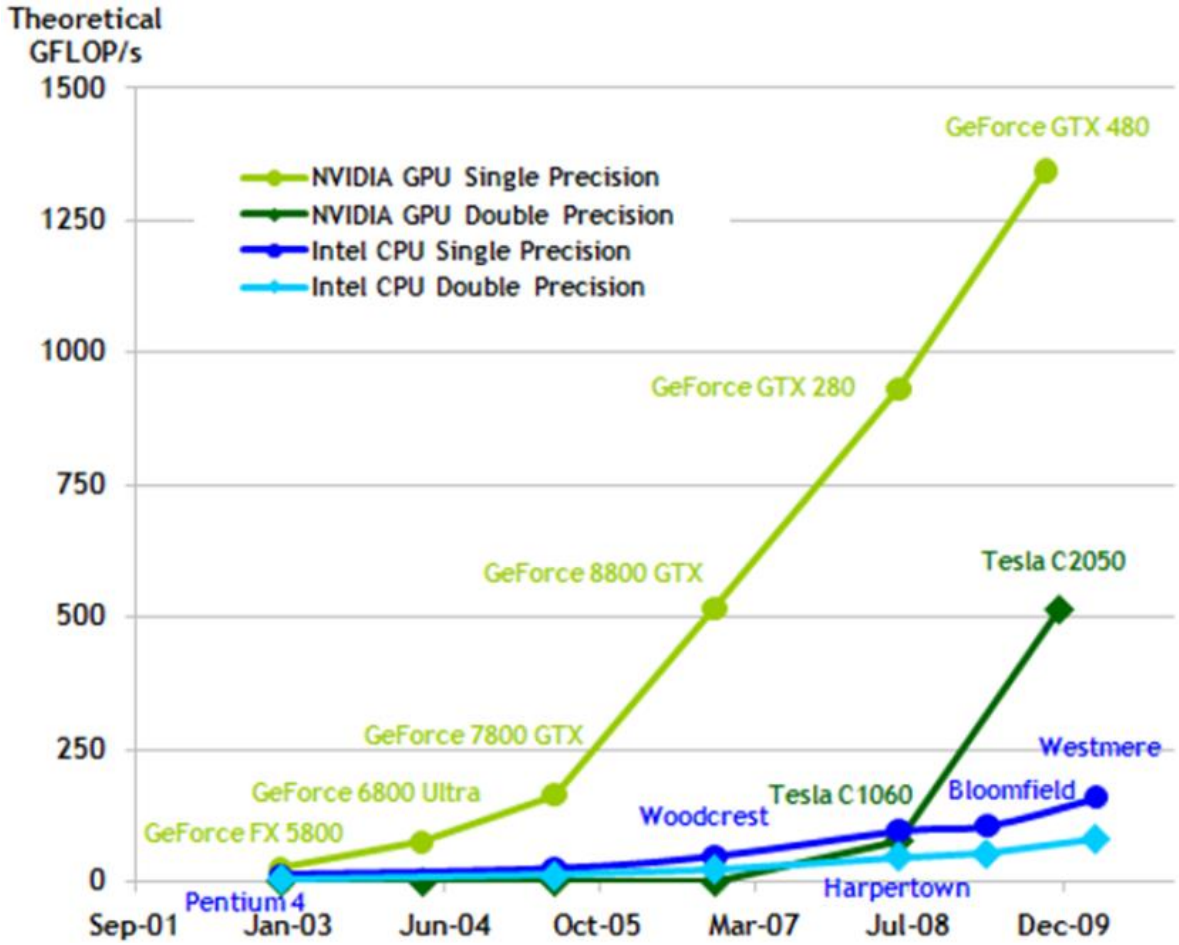
FPGA

- Field-Programmable Gate Array
- Configuration HDL (Hardware-Description Language)

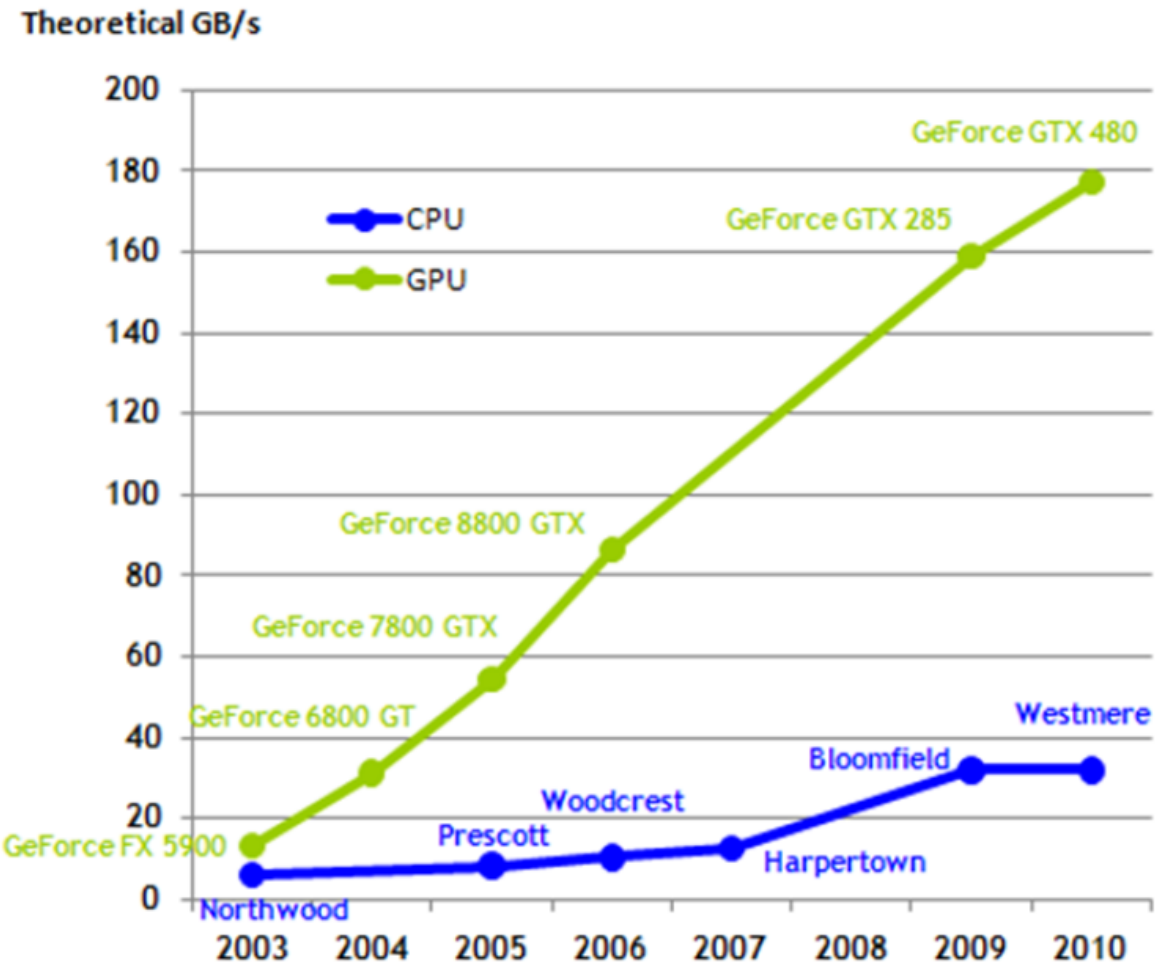


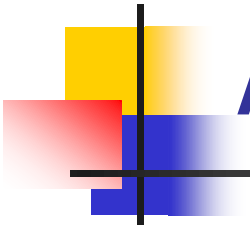
Architecture de cartes graphiques NVIDIA

Evolution architecture NVIDIA – Performances flottantes



Evolution architecture NVIDIA – Bande Passante

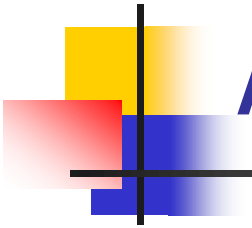




Architecture NVIDIA GeForce

- Brique de base
  - Coeur de calcul très simple
  - Ensemble de coeurs de calcul regroupés dans un *stream multiprocessor*
- Pourquoi *stream* ?
  - Fait pour fonctionner sur un flux de données continu
  - Langage synchrone sur chaque multicoeurs
- Design pour le parallélisme à grain fin

Architecture NVIDIA Kepler



- Dernière génération de la firme NVIDIA (double précision)
  - Nom de code : Kepler
- Nouveautés
  - Augmentation du nombre de cœurs
  - Support des *compute capabilities* 3.5
  - Gestion optimisée du lancement de multiples noyaux de calcul
  - Amélioration des performances relatives en double précision

Architecture Kepler NVidia

- 15 Streaming Multiprocessors (SMs)
- Arrangement perpendiculaire au cache commun L2
- Légende
  - Vert → cœur de calcul
    - Unité double précision en orange
  - Orange → ordonnanceur et *dispatcher*
  - Bleu clair → banc de registres et cache L1

Kepler : *Streaming Multiprocessor*

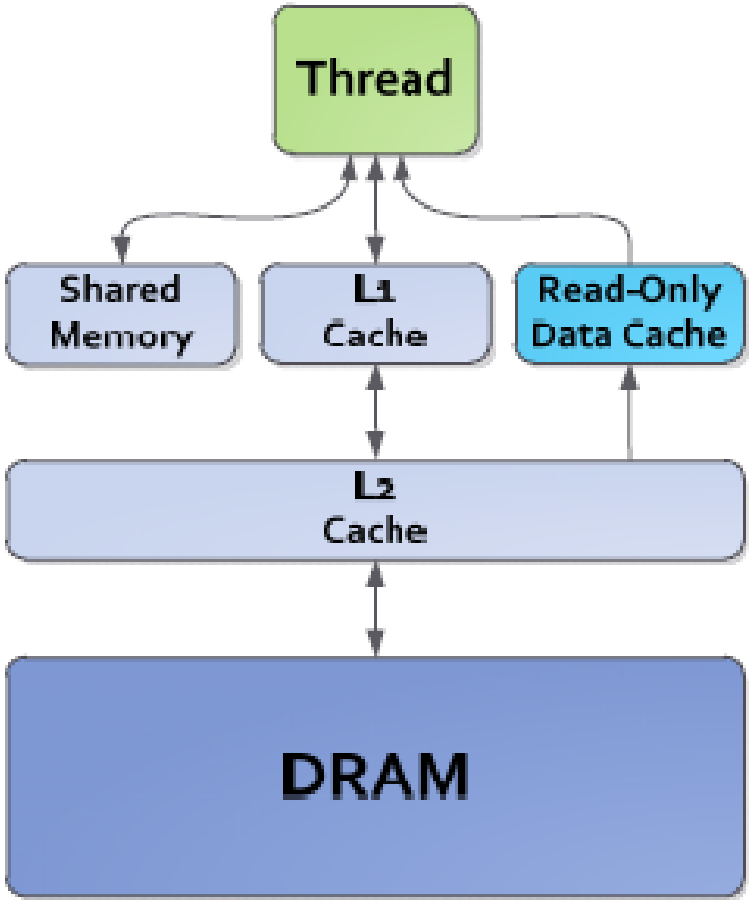
- Streaming Multiprocessor (SMX)
- 192 cœurs compatibles CUDA
  - Unité ALU
  - Unité flottante simple précision
- 64 unités flottantes double précision
- 32 unités spéciales (SFU)
- 32 unités *load/store*

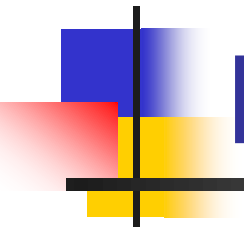
Kepler : *Streaming Multiprocessor*

- Exécution de type SIMT
  - Single-Instruction Multiple Thread
- Similaire au SIMD
- Exécution synchrone des cœurs de calcul
- Chaque SMX gère et exécute les threads de calcul par paquets de 32
  - Notion de *warp*
  - 4 ordonnanceurs de warps par SMX

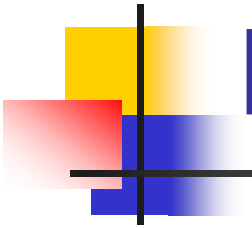
Kepler : hiérarchie mémoire

- Thread ordonnancé sur un coeur de calcul sur un SMX
- Accès mémoire via instruction *load* ou *store*
- Deux possibilités
  - Accès mémoire partagé ➔ pas de cache
  - Accès à la mémoire globale du *device* ➔ passage par les 2 niveaux de cache
- Physiquement, la même mémoire est utilisée pour le cache L1 et la shared memory
- Apparition d'un *read-only cache* (48 Ko)



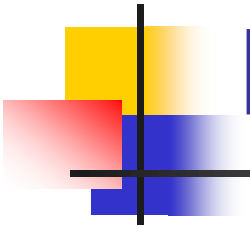


Modèles de programmation pour accélérateurs



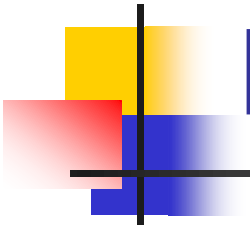
Modèles de programmation

- Langage synchrone et *data flow*
- Langages dédiés
  - Close To Metal (CTM)
  - CAL/IL
  - CUDA
- Langage portables
  - OpenCL
- Directives de parallélisation
  - PGI
  - HMPP (Caps Entreprise)
  - OpenACC



Programmation ATI

- Close to Metal (CTM)
  - Très bas niveau
  - Peu utilisé à présent
- CAL/IL
  - Compute Abstract Layer
    - Interaction entre le processeur principal et l'accélérateur
  - Intermediate Language
    - Langage dédié à l'exécution sur accélérateur
- OpenCL
  - Support du standard OpenCL
  - AMD/ATI fait partie du consortium

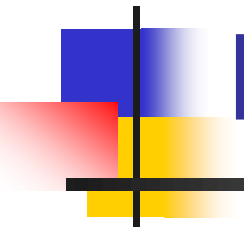


Programmation NVIDIA

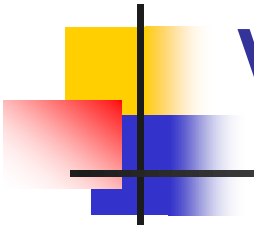
- Langage CUDA
  - Suite…

Directives de parallélisation

- Proposition de programmation à base de directives
  - Ajout de `#pragma` (C,C++) dans le code séquentiel
  - Possibilité d'ajouter ces directives dans un code déjà parallèle (MPI, OpenMP, …) : ceci dépend du modèle
- Solutions proposées
  - OpenACC (*fork* du sous-comité d'OpenMP en charge des accélérateurs)
  - OpenMP 4.5
- Avantages
  - Portabilité en fonction des accélérateurs
  - La sémantique originale (voire séquentielle) peut être préservée
- Inconvénients
  - Souvent besoin de finir *à la main*
  - Spectre parfois limité



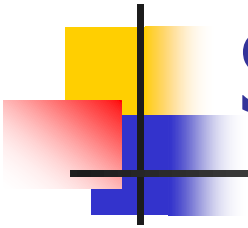
NVIDIA CUDA



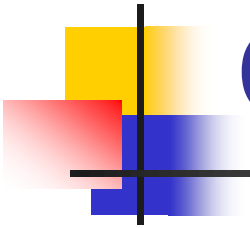
Vue d'ensemble

- Programmation de *kernel*
- Deux parties : programme hôte et noyaux de calcul
- Nécessité de transférer les données
- Chaine de compilation dédiée à cause des noyaux de calcul (langage différent)
- Lien avec un *runtime* (en espace utilisateur) et un *driver* (en espace noyau)

Schéma de programmation



1. Initialisation
   - Allocation mémoire
   - Lecture des entrées
   - …
2. Allocation mémoire sur le GPU
   - Gestion manuelle !
3. Transfert hôte → GPU
4. Exécution d'un noyau de calcul
   - Possibilité de lancer plusieurs noyaux
5. Transferts GPU → hote



Comment compiler ?

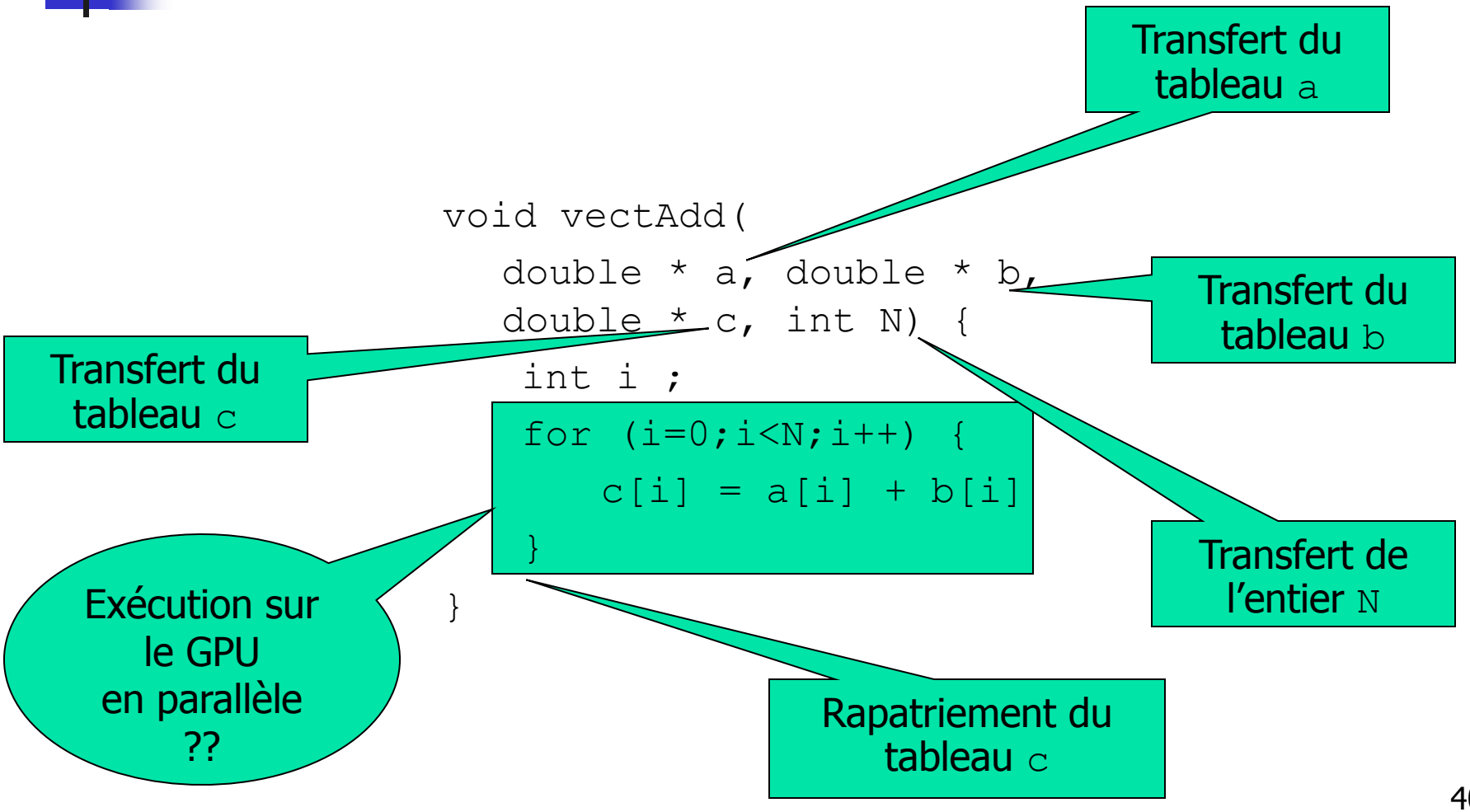
- Structure du code
  - Partie pour l'hôte
  - Partie pour le *device*
- En pratique, ces codes peuvent êtres mélangés dans le même fichier
- Programmation hôte
  - C ou C++
- Programmation *device*
  - CUDA (sur-ensemble du C norme 99)
- Chaîne de compilation : NVCC
  - En tant que programmeur : utilisation du compilateur `nvcc`
  - N'hésitez pas à regarder la documentation livrée avec le SDK

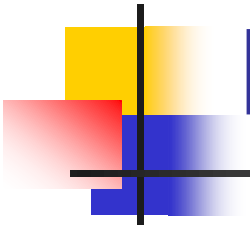
Exemple de programme

- Exemple : addition de 2 vecteurs
- Opération de base :
  - c[i] = a[i]+b[i]
- Comment programmer ce noyau de calcul en séquentiel ?
  - Simple boucle
  - Application de l'opération de base sur chaque élément

```
void vectAdd(
   double * a, double * b,
   double * c, int N) {
    int i ;
    for (i=0;i<N;i++) {
        c[i] = a[i] + b[i]
    }
}
```

Portage de notre exemple

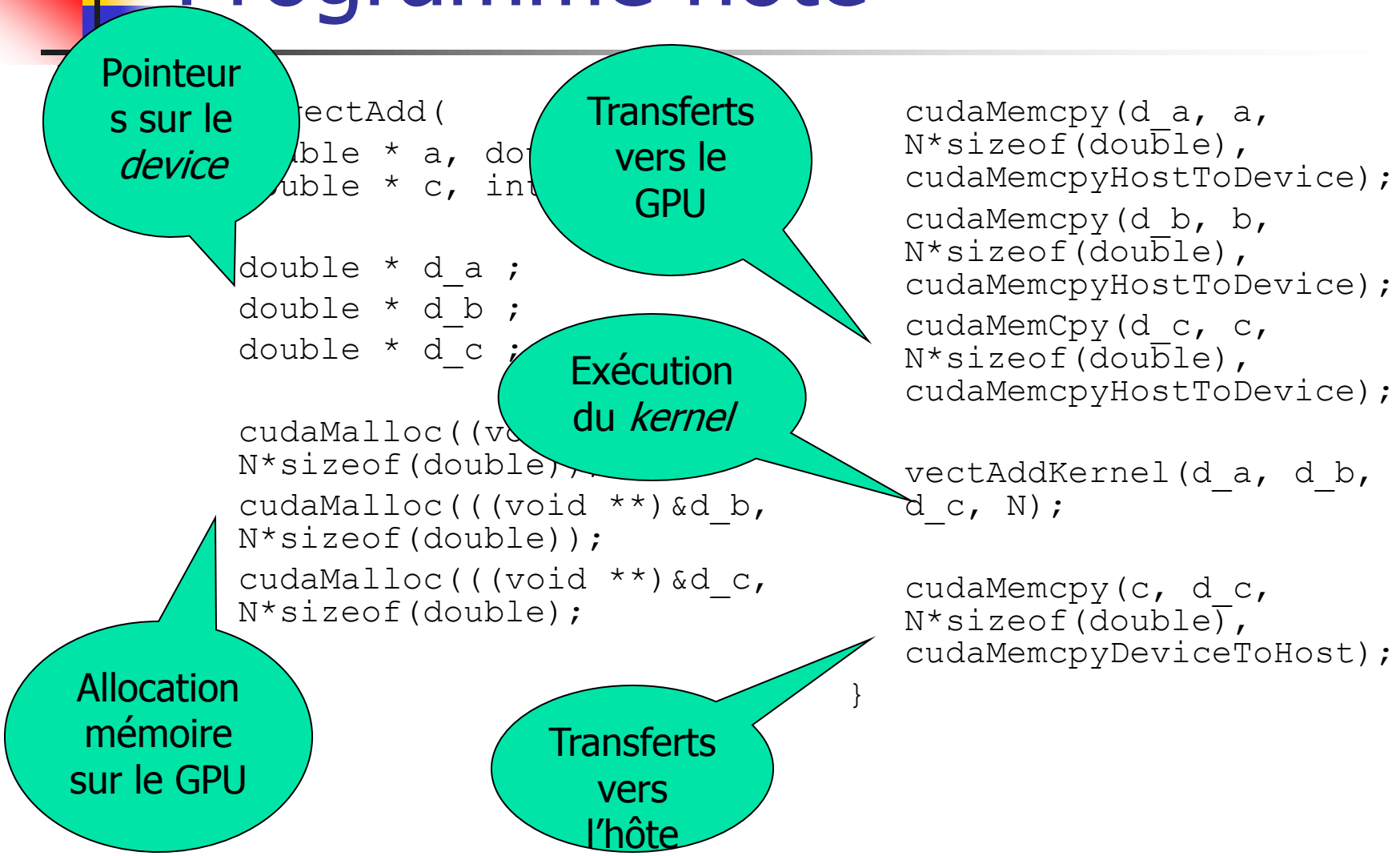




Programmation sur l'hote

- Gestion de l'hôte (CPU central)
  - Besoin de gérer la mémoire du GPU
  - Besoin de transférer les données
  - Donne l'ordre d'exécution du kernel
- API CUDA de l'hôte
  - CUDA Haut niveau / Bas niveau
  - `cudaMalloc` : allouer des données sur le *device*
  - `cudaMemcpy` : transférer des données vers le *device* ou vers l'hôte

Programme hôte




```
ectAdd(
ble * a, do
uble * c, int

double * d_a ;
double * d_b ;
double * d_c ;

cudaMalloc((v
N*sizeof(double));
cudaMalloc((void **)&d_b,
N*sizeof(double));
cudaMalloc((void **)&d_c,
N*sizeof(double);

}
```

```
cudaMemcpy(d_a, a,
N*sizeof(double),
cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b,
N*sizeof(double),
cudaMemcpyHostToDevice);
cudaMemCpy(d_c, c,
N*sizeof(double),
cudaMemcpyHostToDevice);

vectAddKernel(d_a, d_b,
d_c, N);

cudaMemcpy(c, d_c,
N*sizeof(double),
cudaMemcpyDeviceToHost);
```

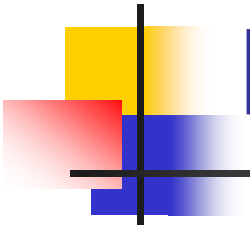
Modèle d'exécution

- Le *device* n'a plus qu'à exécuter le code en lui-même : l'addition de 2 vecteurs
- Comment programmer sur le *device* ?

```
for (i=0;i<N;i++)
   c[i]=a[i]+b[i];
```

- Le modèle CUDA définit la notion de thread
  - Chaque thread lance la fonction *kernel*
- La programmation CUDA est alors par défaut *multithreadée*
  - Il faut raisonner par thread et non pas par boucle ou par itération de boucles
  - Différent d'une programmation séquentielle classique
- Code du *kernel* dans notre cas :

```
int i ;
/* Update i with thread index */
c[i] = a[i] + b[i];
```

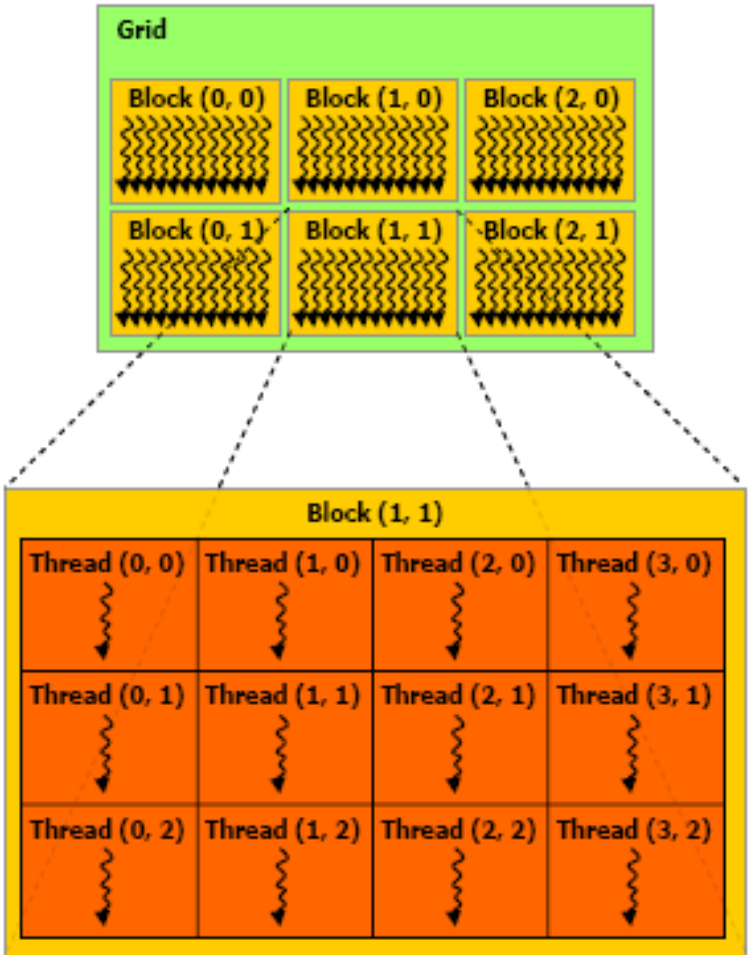


Hiérarchie de threads

- Comment connaitre l'identifiant d'un thread ?
  - Ceci est lié à l'organisation des threads au niveau du modèle d'exécution
- CUDA définit la notion de bloc *(block)* et grille *(grid)* pour organiser les threads de façon hiérarchique
- Une grille
  - Contient des blocs
    - Qui contiennent des threads
- La grille et les blocs peuvent avoir de 1 à 3 dimensions (représentation logique et non physique)
- Concrètement ?

Hiérarchie de threads

- Exemple avec
  - 1 grille à 2 dimensions
  - 6 blocs à 2 dimensions
- Il y a 6 blocs dans la grille et 12 threads par bloc
- ➔ 72 threads au total
- Calcul d'un indice de thread ?



Programmation CUDA

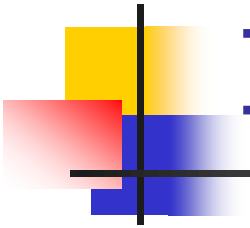
- Déclaration d'un kernel
  - Mot clé __global__
- Calcul de l'indice du thread en fonction de la géométrie de la grille
  - Exemple avec une grille à 1 dimension et des blocs à 1 dimensions
- Test pour s'assurer que les threads n'écrivent pas en dehors du tableau
  - Correspond à la borne de la boucle d'origine

```
__global__ void
   vecAddKernel( double *a,
   double *b, double *c) {

   int i ;

   i = blockIdx.x *
   blockDim.x + threadIdx.x
   ;

   if ( i<N ) {
        c[i] = a[i]+b[i];
   }
}
```



Intégration des *kernels*

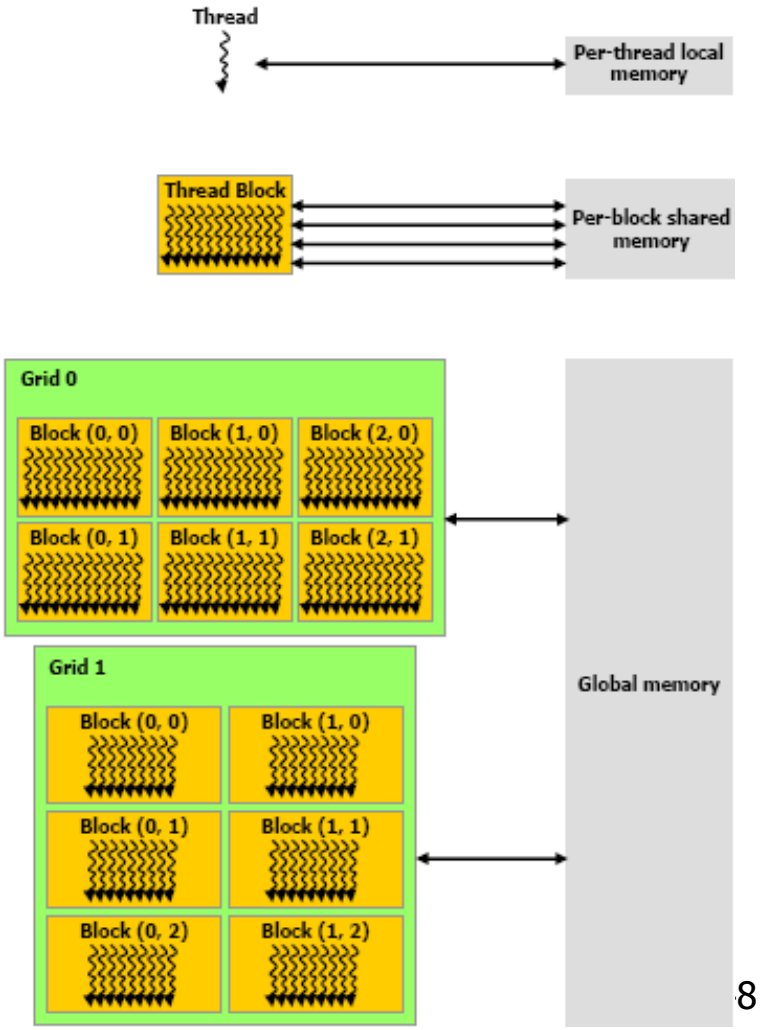
- Comment se forment les blocs et la grille ?
- Besoin de le préciser lors de l'exécution du kernel
  - Le programme hôte est en charge de définir les dimensions du calcul
- Syntaxe d'un appel à un noyau de calcul du *device*

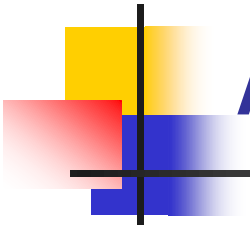
```
my_kernel<<<Dg, Db>>>(arg1, arg2, arg3 );
```

  - Dg : dimensions et taille de la grille (type dim3)
  - Db : dimensions et taille des blocs de la grille (type dim3)
- Nombre total de blocs ?
- Nombre de threads par bloc ?
- Nombre total de threads ?

Hiérarchie mémoire

- Vision par entité de calcul
- Thread
  - Accès mémoire local
  - Banc de registres privé
- Bloc
  - Shared memory privé
- Grille
  - Mémoire globale



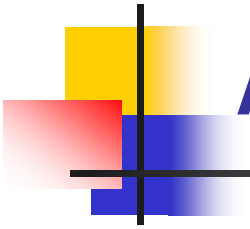


Asynchronisme

- Par défault certaines fonctions *rendent la main* au programme hôte
  - Exécution de kernel
  - Copies *device* vers *device*
  - Initialisation de la mémoire
- Possibilité d'attendre la fin de l'exécution à un instant donné

```
cudaThreadSynchronize();
```

- Appels consécutifs à notre noyau *vecAdd*
  - Si les vecteurs sont différents ?
  - S'il existe une dépendance RAW dans notre calcul ?



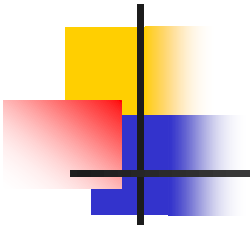
Asynchronisme

- Même si la main est rendue à l'hôte, la sémantique reste séquentielle
  - Les noyaux de calcul sont exécutés dans l'ordre
  - La gestion des dépendances est implicites
- Comment obtenir du *vrai* asynchronisme ?

Asynchronisme avancé

- Motivations :
  - Pouvoir transférer des données pendant l'exécution d'un kernel
  - Pouvoir exécuter plusieurs noyaux de calculs
    - Si la carte graphique le permet (Fermi)
    - S'il n'y a pas de dépendances entre les noyaux
- Solution : *streaming*

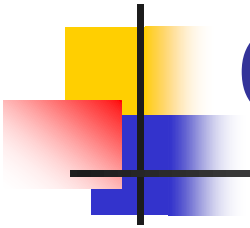
- Déclaration d'un ou plusieurs *streams*
  - Chaque interaction avec le *device* se fait sur un stream en particulier
  - Le driver sait alors ce qui peut être parallélisé ou non
  - `cudaMemCpy(d_c, c, N*sizeof(double), cudaMemcpyHostToDevice, stream[0]);`
  - `my_kernel<<<Dg,Db,stream[0]>>>(arg1, arg2, arg3);`



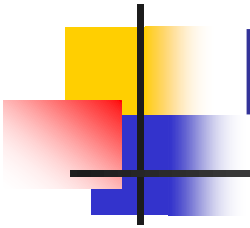
Timing et suivi du programme

- Mesure de temps *(profiling)*
  - Utilisation des évenements définis par CUDA
- Type principal : `cudaEvent_t`
- Création
  - `cudaEventCreate(cudaEvent_t * e)`
- Activation
  - `cudaEventRecord(cudaEvent_t e, cudaStream_t s)`
- Attente de l'activation des évênements
  - `cudaEventSynchronize(cudaEvent_t e);`
- Calcul du temps passé
  - `cudaEventElapsedTime(float * ms, cudaEvent_t start, cudaEvent_t stop);`

Gestion des erreurs



- En CUDA, (presque) toutes les fonctions retournent un code d'erreur
  - Retour d'un type `cudaError_t`
- Si tout se passe bien, il s'agit alors de `cudaSuccess`
- Sinon, il est possible d'obtenir une description
  - `const char * cudaGetErrorString (cudaError_t error);`
- Pour les fonctions ne retournant pas une telle info (par exemple, un appel à un *kernel*)
  - Appel à `cudaGetLastError()`
  - Retourne un type `cudaError_t`



Règles de programmation

- Nous avons vu
  - Gestion du GPU à partir du CPU (hôte)
  - Interaction entre l'hôte et le *device*
- Mais, comment débugger un code ?
  - Utilisation de *printf* directement possible mais pas infaillible
  - Ajout de synchronisation
  - Vérifier le retour de chaque fonction CUDA