

# Architecture et programmation d'accélérateurs matériels

Cours 1 : Architecture et modèles de  
programmation

Julien Jaeger  
[julien.jaeger@cea.fr](mailto: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
  - Introduction à CUDA
- Intervenant
  - Julien Jaeger ([julien.jaeger@cea.fr](mailto:julien.jaeger@cea.fr))
  - Adrien Roussel ([adrien.roussel@cea.fr](mailto:adrien.roussel@cea.fr))

# Plan du cours

- Panorama des accélérateurs
- Architecture de GPGPU
  - NVIDIA Kepler
  - NVIDIA Volta
- 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 contrôlée 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 *worker*.
- 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

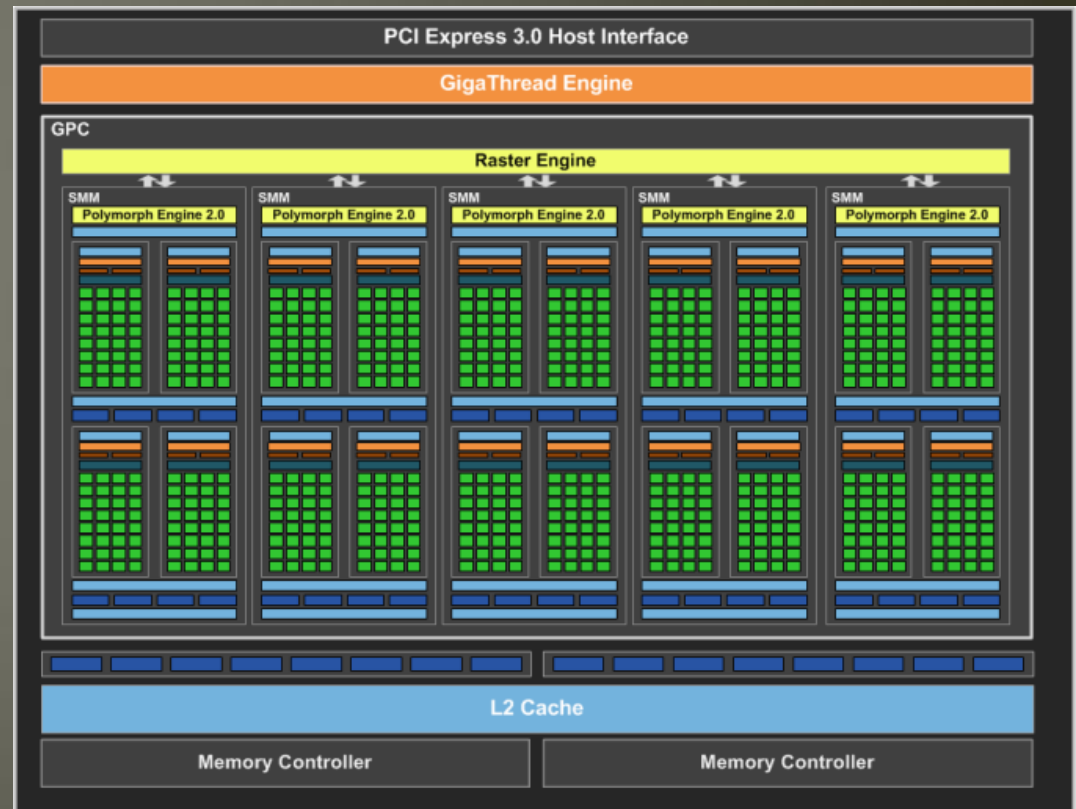
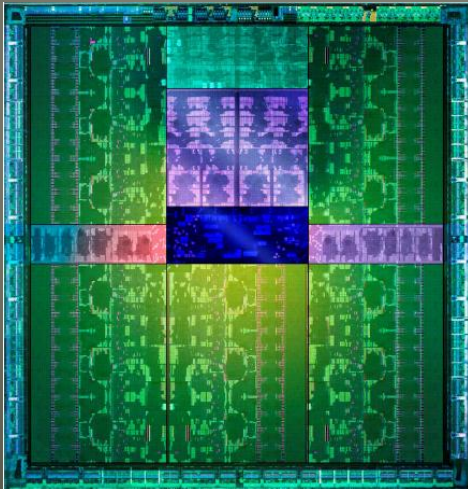
# Processeurs graphiques

- 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 (General Purpose Graphics Processing Unit)

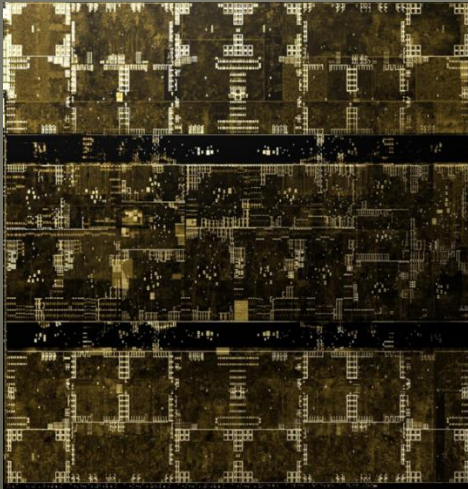
# GPGPU

- 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

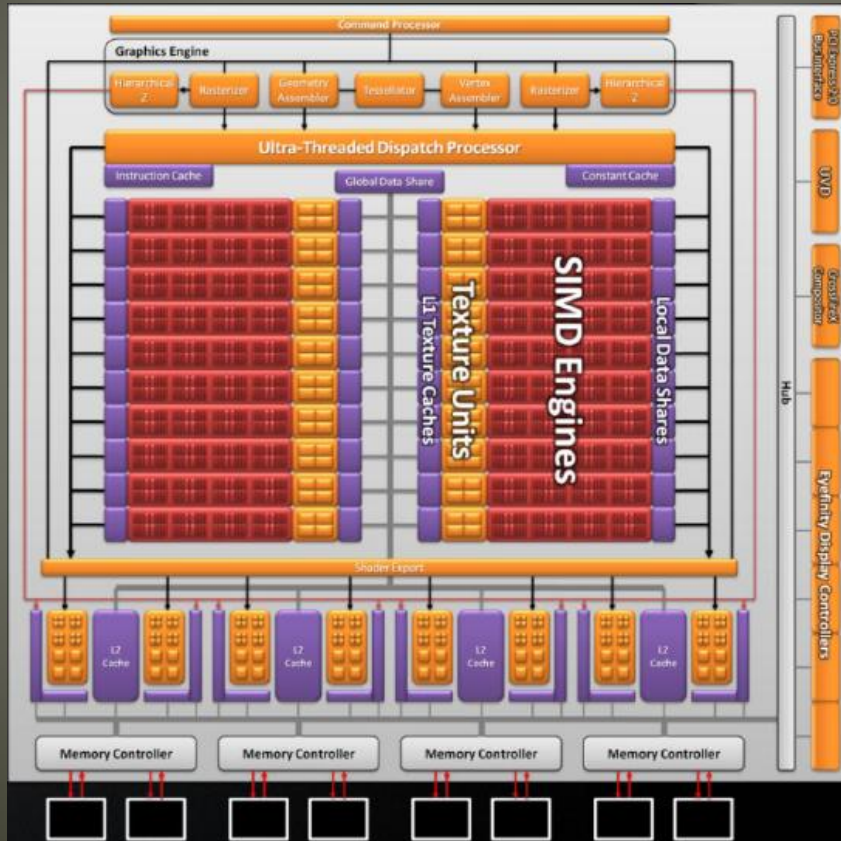


# Volta



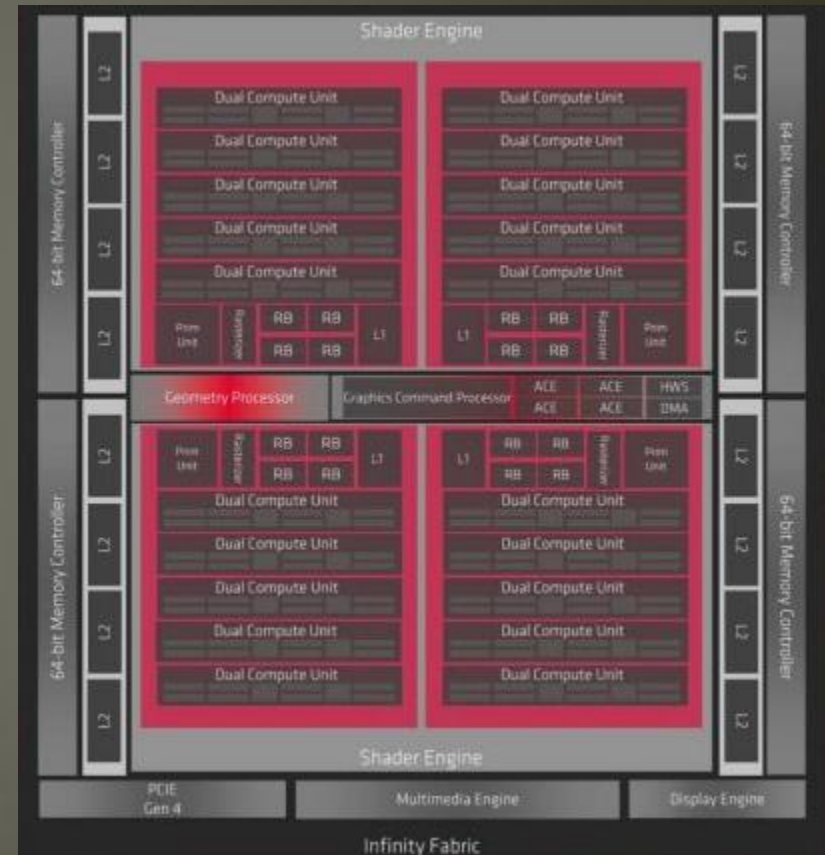
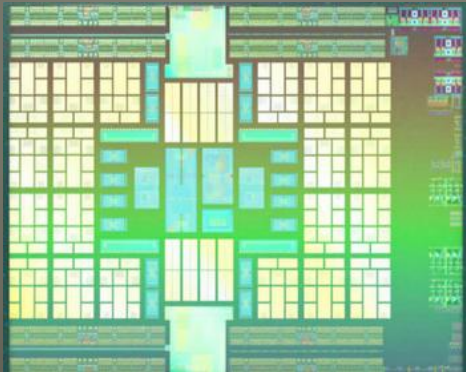


# Architecture ATI Cypress

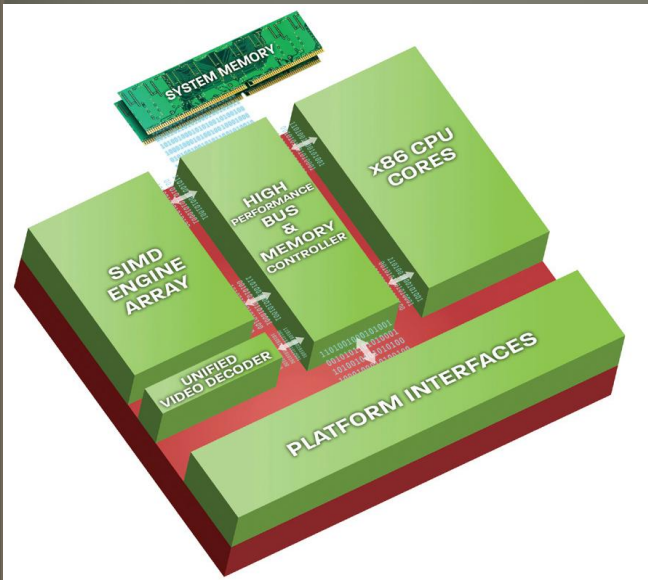


- Ensemble d'unités vectorielles
  - SIMD Engines
- Hiérarchie mémoire
  - Cache L2
- Performances crêtes
  - 2,72 TFlops SP
  - 0.54 TFlops DP

# Architecture ATI Radeon Navi



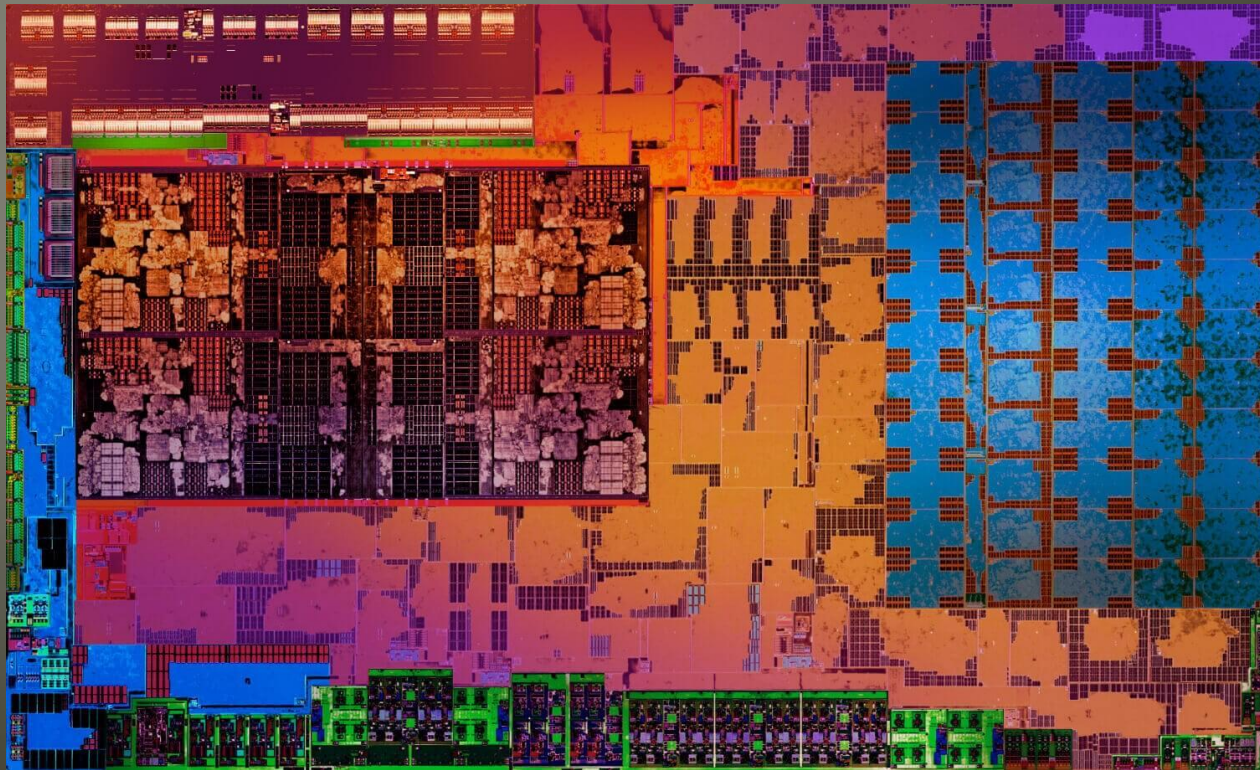
# Architecture Fusion (Llano)



- Nom de code de l'architecture d'AMD/ATI : Fusion
- Design d'un APU (Accelerator Processing Unit)
- Sur la même puce :
  - CPU
  - Accélérateur de type GPU
- Première génération disponible en 2011
  - Llano avec technologie Bulldozer ou Bobcat
- <http://fusion.amd.com>

# Architecture Fusion (Picasso)

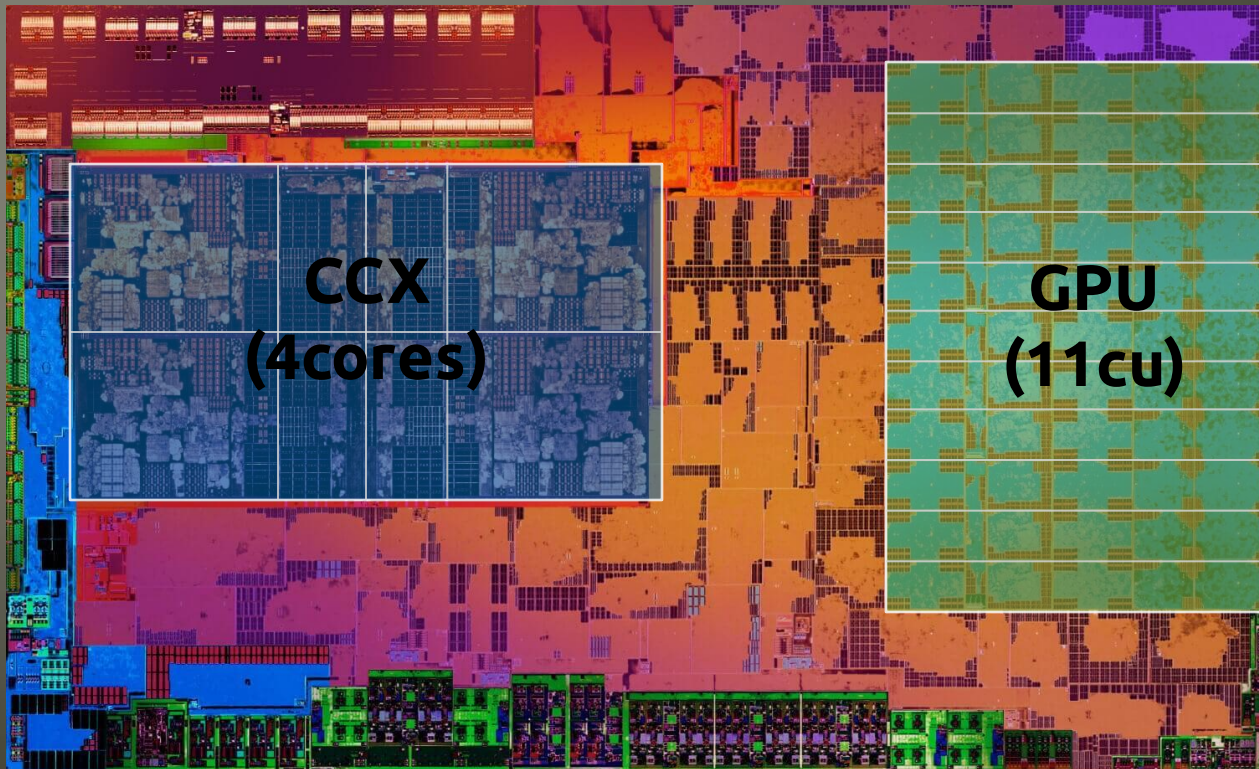
- Ryzen-bridge 2019



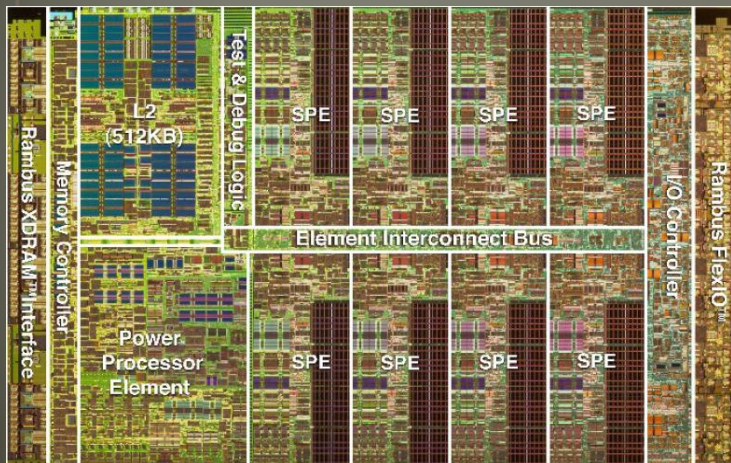


# Architecture Fusion (Picasso)

- Ryzen-bridge 2019



# Le processeur CELL (IBM)



- Développé conjointement par 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.

# 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
- Première version: KNF
- Version “actuelle” : KNL

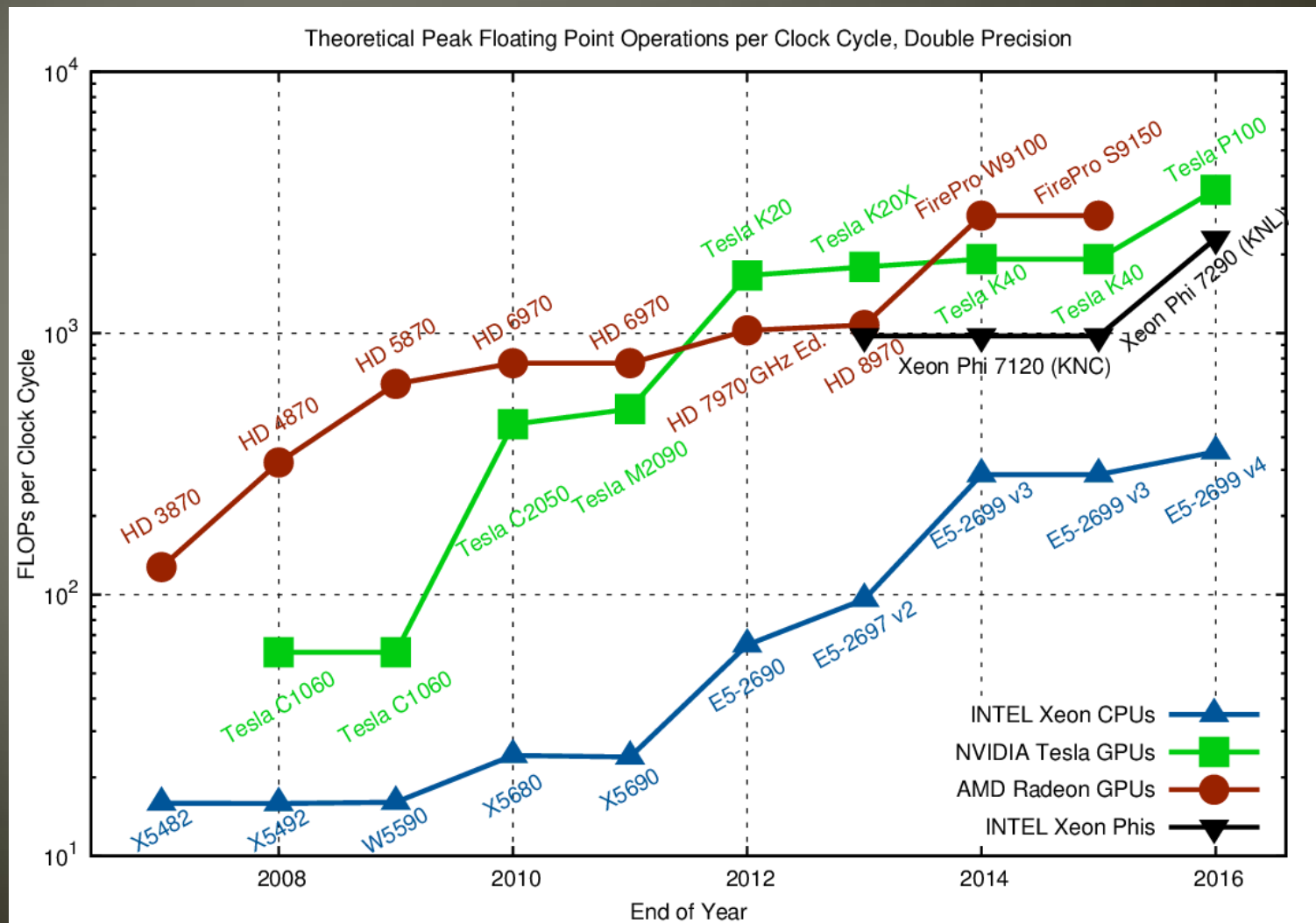
# FPGA

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



# Architecture de cartes graphiques NVIDIA

# Evolution architecture NVIDIA – Performances flottantes



# 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

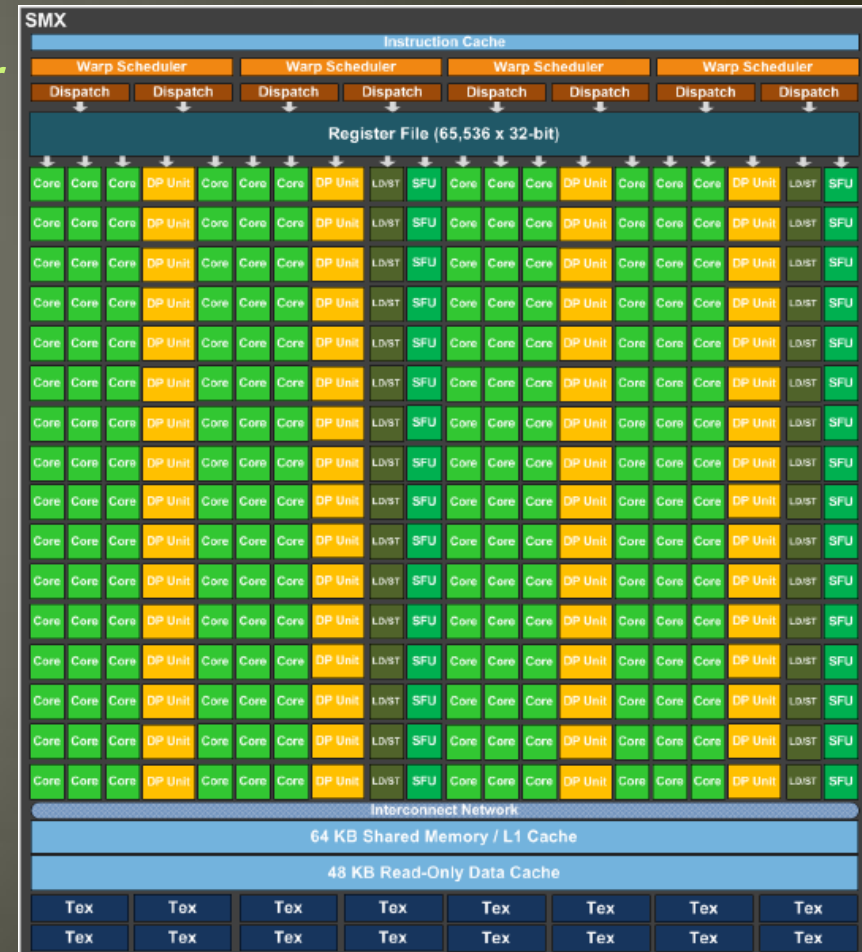
# Architecture Kepler NVidia

- 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



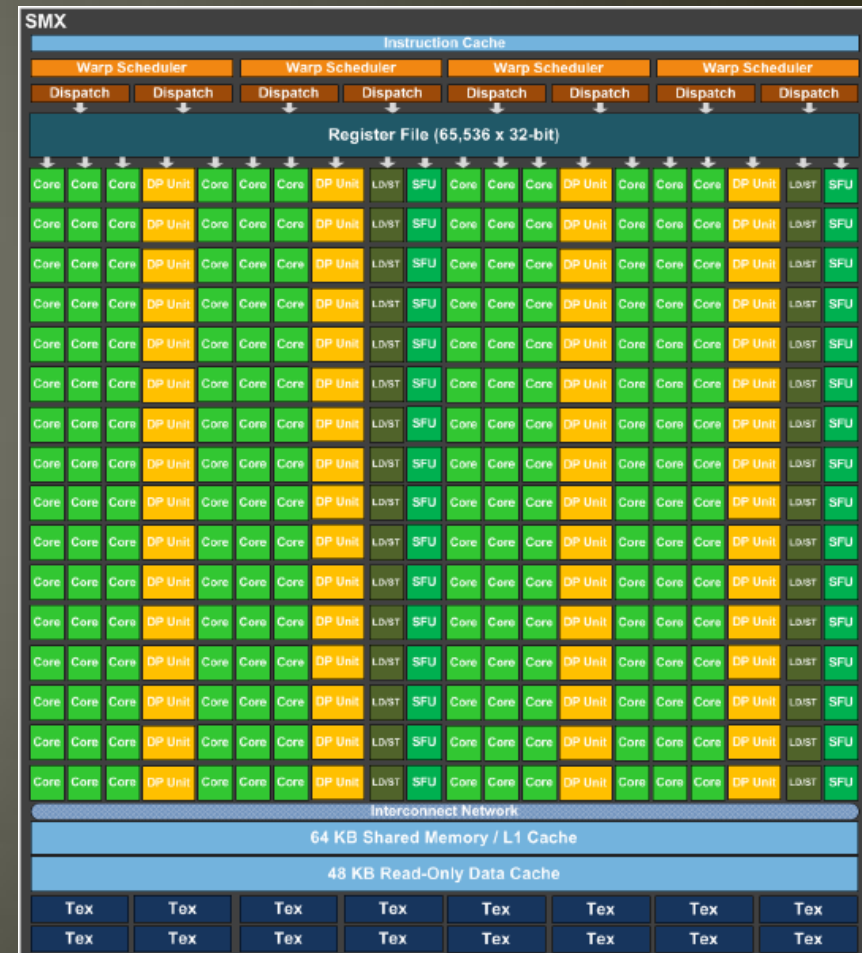
# Kepler : Streaming Multiprocessor

- 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*



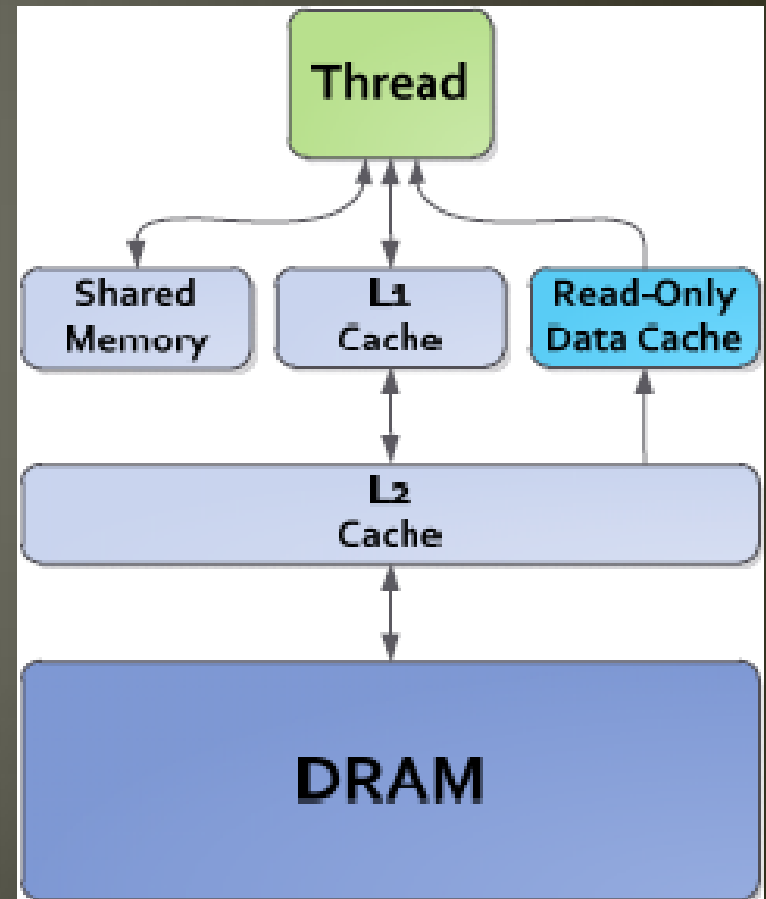
# Kepler : Streaming Multiprocessor

- 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)





# Volta



- 6 GPCs (Gpus Processing clusters)
- 7 TPCs (Texture Processing clusters) / GPC
- 2 SM par TPCs
- $\rightarrow 2 * 7 * 6 = 84 \text{ SM!}$



# Volta



- 84 Streaming Multiprocessor
- 5376 unités flottantes simple précision (14 Tflops)
- 5376 unités entières simple précision (14 Tflops)
- 2688 unités flottantes double précision (7 Tflops)
- 672 cœurs tenseur (112 Tflops)
- 336 unités de texture

# 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
- **ROCm**
  - Offers toolkit to write and translate CUDA codes
- **OpenCL/SYCL**
  - Support des standards OpenCL et SYCL
  - 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
  - OpenMP 4+
- 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 → hôte

## 6. Libération mémoire GPU

## 7. Libération mémoire CPU

# Comment compiler ?

- **Structure du code**
  - Partie pour l'hôte
  - Partie pour le *device*
- **En pratique, ces codes peuvent être 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

```
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

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

Transfert du  
tableau c

Transfert du  
tableau a

Transfert du  
tableau b

Transfert de  
l'entier N

# Portage de notre exemple

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

Transfert du  
tableau a

Transfert du  
tableau b

Transfert du  
tableau c

Transfert de  
l'entier N

Exécution sur  
le GPU  
en parallèle  
??

# Portage de notre exemple

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

Transfert du  
tableau a

Transfert du  
tableau b

Transfert du  
tableau c

Transfert de  
l'entier N

Exécution sur  
le GPU  
en parallèle  
??

Rapatriement du  
tableau c



# Programmation sur l'hôte

- 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

```
void vectAdd(
    double * a, double * b,
    double * c, int N) {

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

    cudaMalloc((void **)&d_a,
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);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

# Programme hôte

Pointeurs  
sur le  
*device*

```
vectAdd(
    double * a, double * b,
    double * c, int N) {
    double * d_a ;
    double * d_b ;
    double * d_c ;

    cudaMalloc((void **)&d_a,
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);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

```
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);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

}

# Programme hôte

Pointeurs  
sur le  
device

```
void vectAdd(
double * a, double * b,
double * c, int N) {
double * d_a ;
double * d_b ;
double * d_c ;

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

Allocation  
mémoire  
sur le GPU

```
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);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
}
```

# Programme hôte

Pointeurs  
sur le  
device

```
void vectAdd(
double * a, double * b,
double * c, int N) {
double * d_a ;
double * d_b ;
double * d_c ;

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

Allocation  
mémoire  
sur le GPU

Transferts  
vers le  
GPU

```
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);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
}
```

# Programme hôte

Pointeurs  
sur le  
*device*

```
void vectAdd(  
double * a, double * b,  
double * c, int N) {  
double * d_a ;  
double * d_b ;  
double * d_c ;
```

Exécution  
du *kernel*

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

Allocation  
mémoire  
sur le GPU

Transferts  
vers le  
GPU

```
    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);
```

```
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);
```

```
}
```

# Programme hôte

Pointeurs  
sur le  
*device*

```
void vectAdd(  
double * a, double * b,  
double * c, int N) {  
double * d_a ;  
double * d_b ;  
double * d_c ;
```

Exécution  
du *kernel*

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

Allocation  
mémoire  
sur le GPU

Transferts  
vers l'hôte

```
    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);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

Transferts  
vers le  
GPU



# Programme hôte

Pointeurs  
sur le  
*device*

```
void vectAdd(  
double * a, double * b,  
double * c, int N) {  
double * d_a ;  
double * d_b ;  
double * d_c ;
```

Exécution  
du *kernel*

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

Allocation  
mémoire  
sur le GPU

Transferts  
vers l'hôte

```
    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);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

Transferts  
vers le  
GPU

Libération  
mémoire  
sur le GPU

# 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*
    - Un peu comme si de nombreux threads étaient créés avec `pthread_create` en passant la fonction kernel en fonction à exécuter
- 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 :

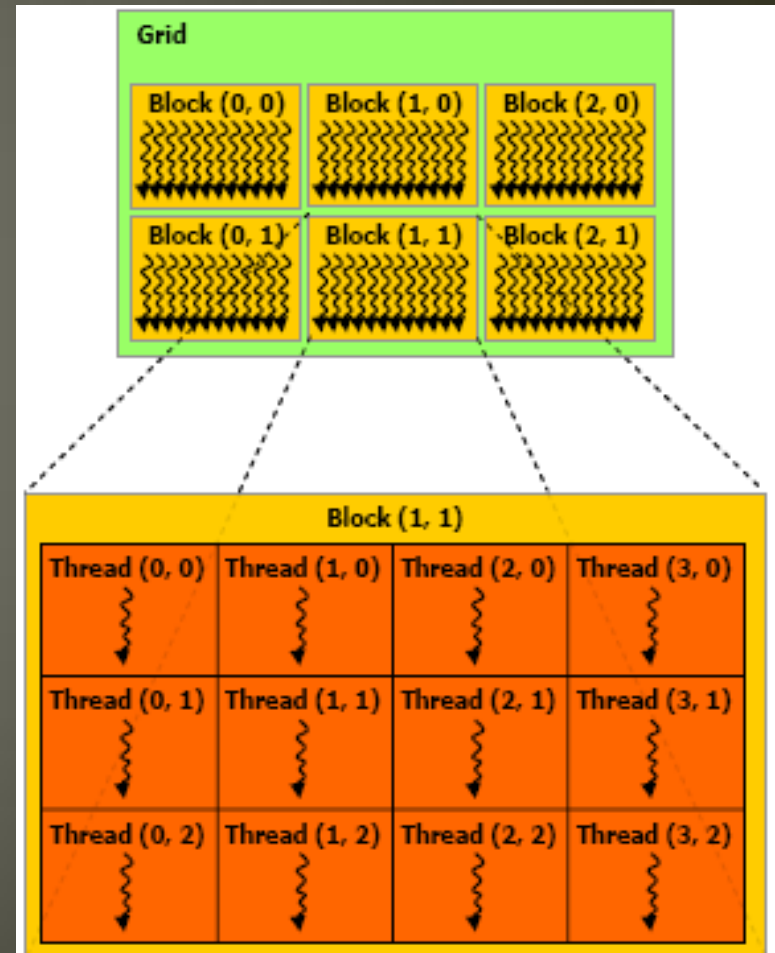
```
int i ;  
i= index; /* Update i with thread index */  
c[i] = a[i] + b[i];
```

# Hiérarchie de threads

- Comment connaître 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 ?

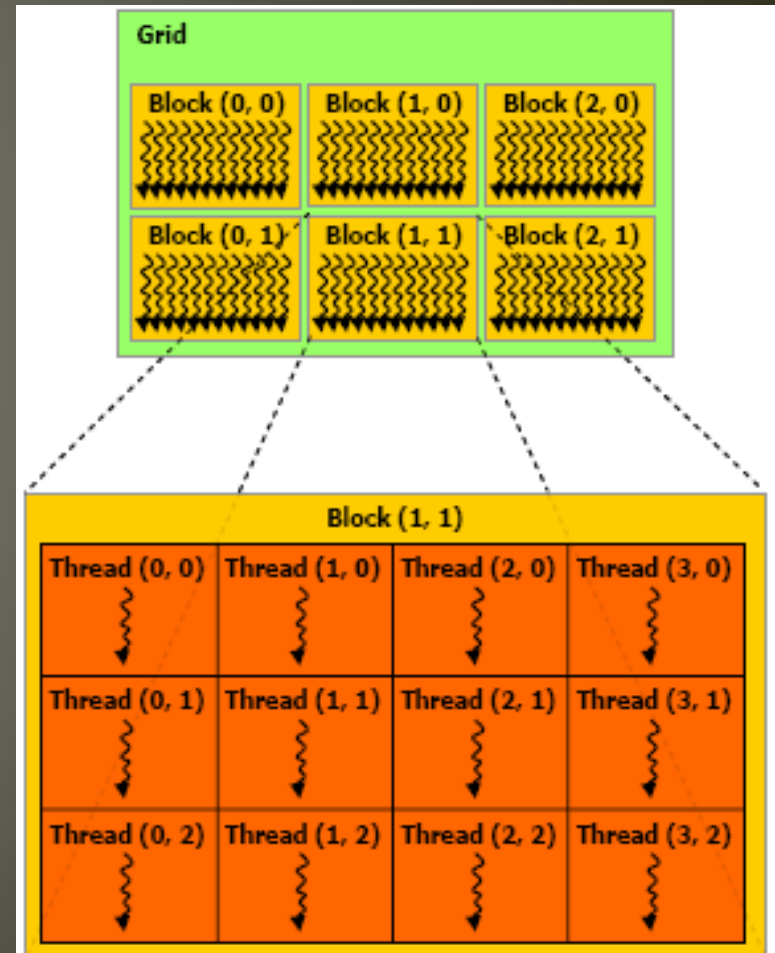
# Hiérarchie de threads

- 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 ?
  - Linéarisation de tableau à n-D



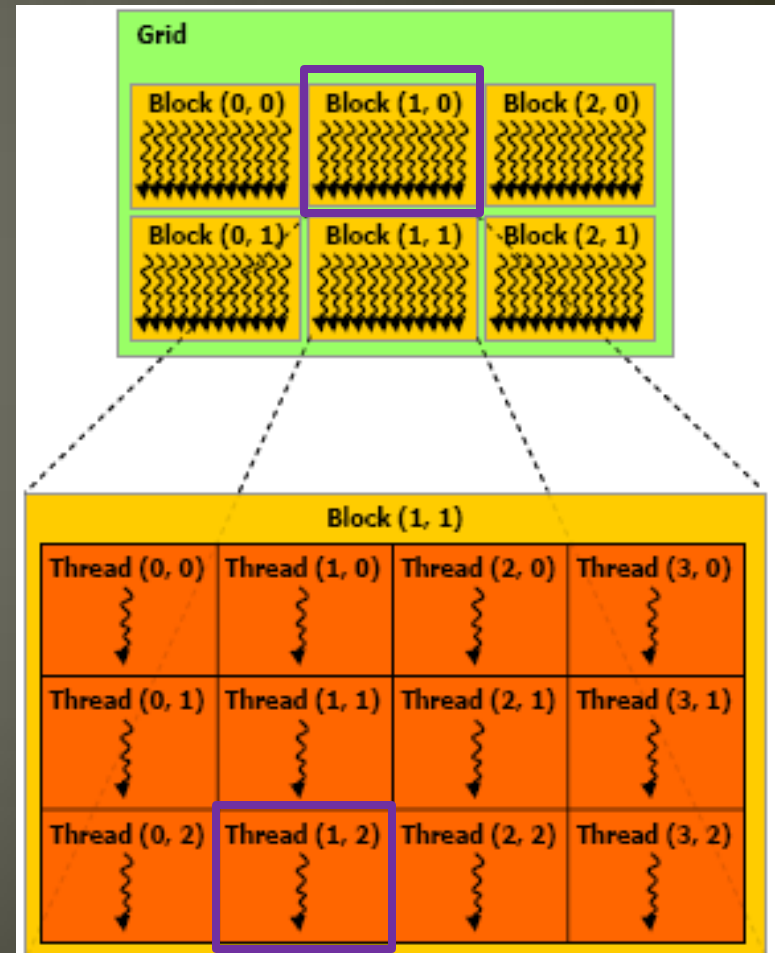
# Hiérarchie de threads

- **Dimensions de la grille**
  - `dim3 gridDim`
  - Contient le nombre de blocs dans les 3 dimensions de la grille
- **Indice d'un bloc dans la grille**
  - `dim3 blockIdx`
  - Contient les coordonnées du bloc courant dans la grille
- **Dimensions des blocs**
  - `dim3 blockDim`
  - Contient le nombre de threads dans les 3 dimensions des blocs
- **Indice du thread dans le bloc**
  - `uint3 threadIdx`
  - Contient l'indice du thread courant dans le bloc courant (3 dimensions)
- **Taille d'un warp**
  - `int warpSize`



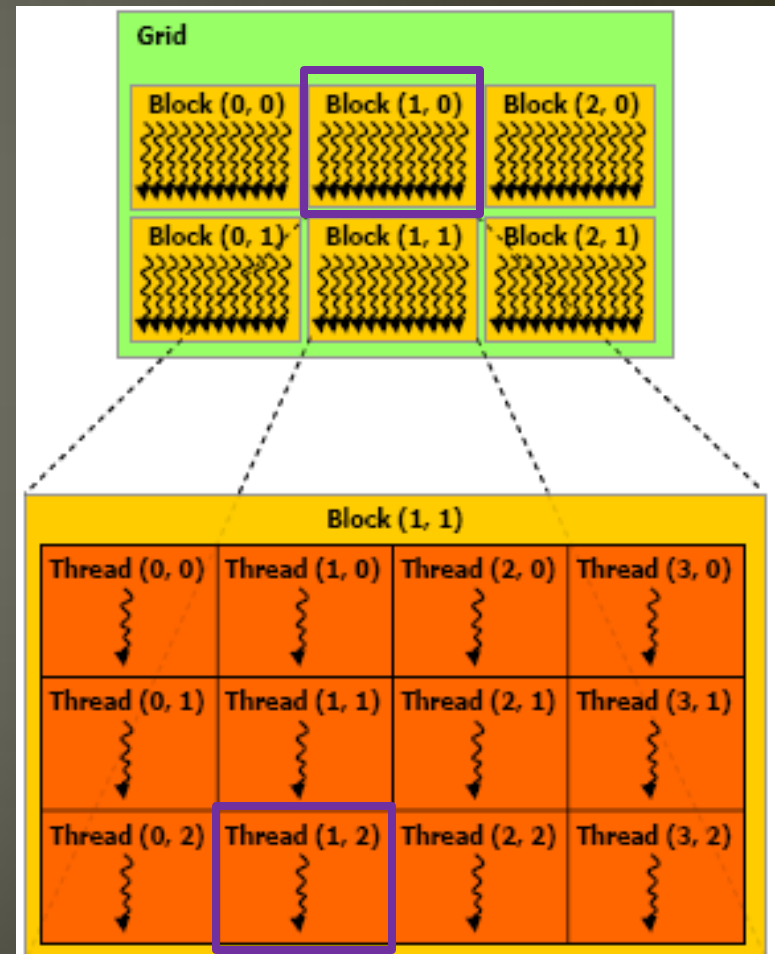
# Hiérarchie de threads

- **Dimensions de la grille**
  - `dim3 gridDim`
  - Contient le nombre de blocs dans les 3 dimensions de la grille
- **Indice d'un bloc dans la grille**
  - `dim3 blockIdx`
  - Contient les coordonnées du bloc courant dans la grille
- **Dimensions des blocs**
  - `dim3 blockDim`
  - Contient le nombre de threads dans les 3 dimensions des blocs
- **Indice du thread dans le bloc**
  - `dim3 threadIdx`
  - Contient l'indice du thread courant dans le bloc courant (3 dimensions)
- **Taille d'un warp**
  - `int warpSize`



# Hiérarchie de threads

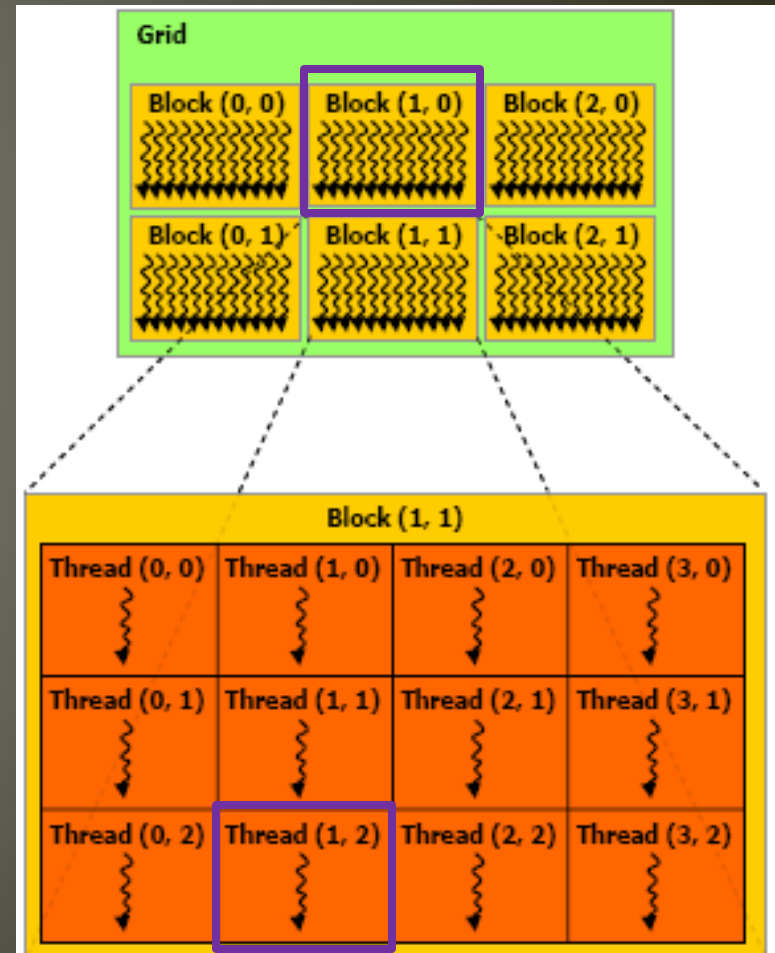
- Calculons l'ID global du thread encadré





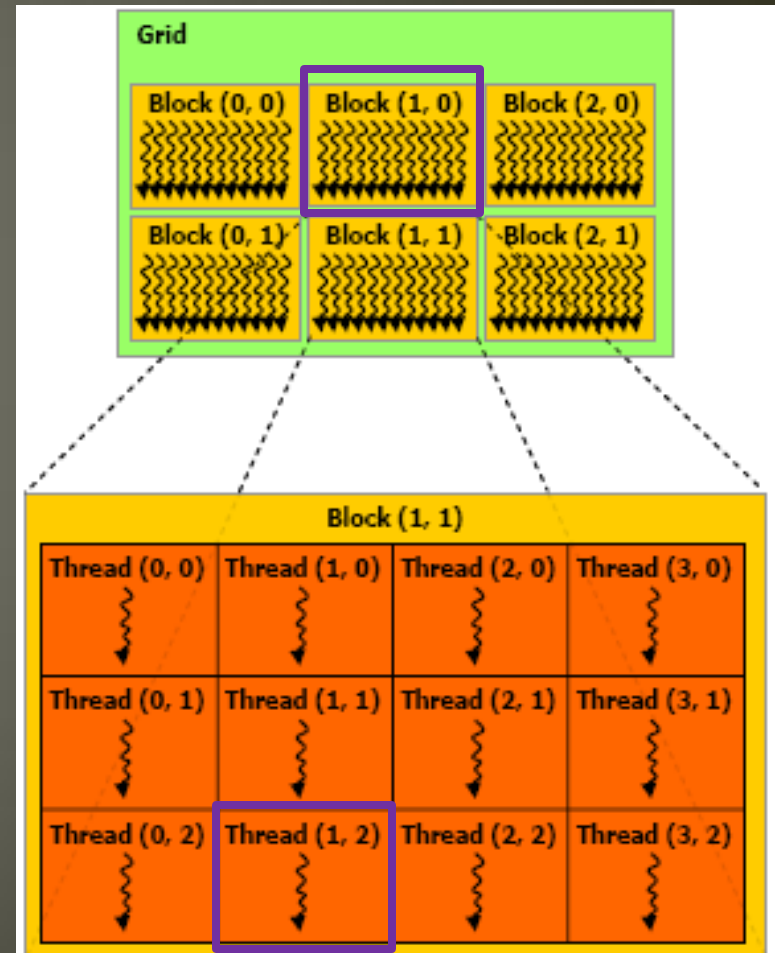
# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$



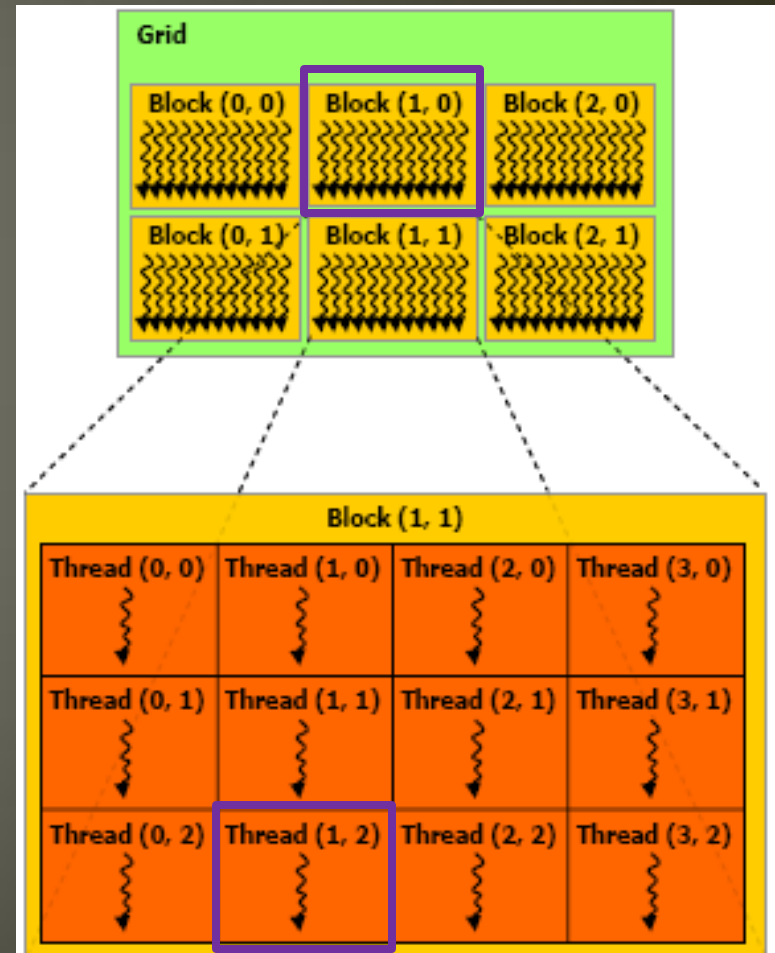
# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$
- $\text{Block\_glob\_id} = \text{BlockIdx.y} \times \text{GridDim.x} + \text{BlockIdx.x} = 0 \times 3 + 1 = 1$



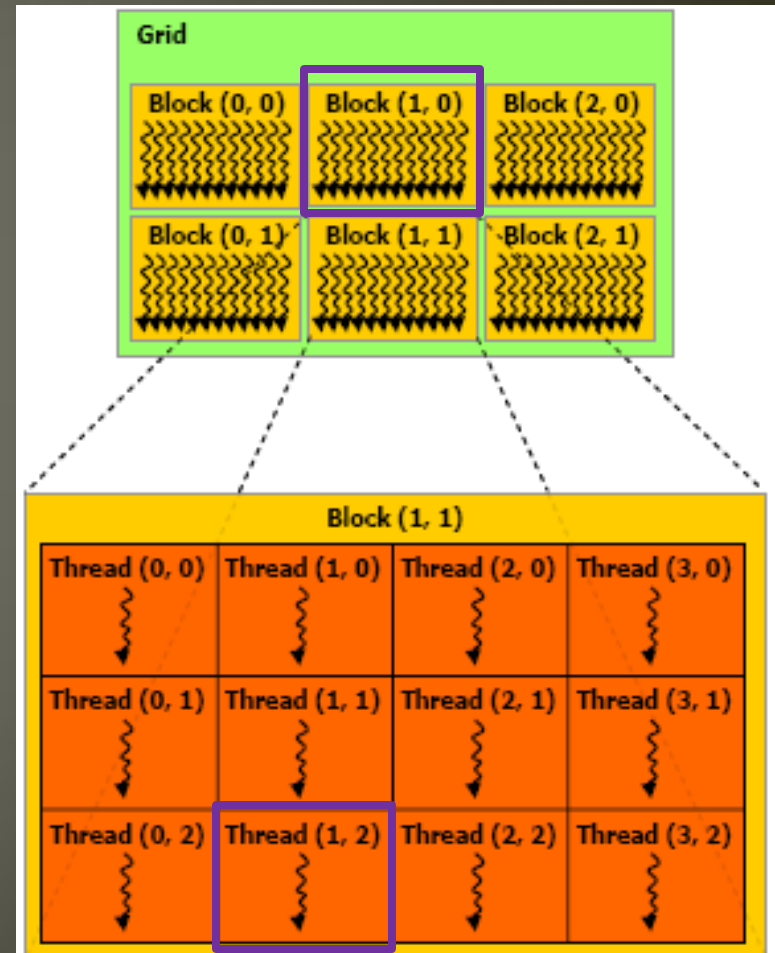
# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$
- $\text{Block\_glob\_id} = \text{BlockIdx.y} \times \text{GridDim.x} + \text{BlockIdx.x} = 0 \times 3 + 1 = 1$ 
  - $\text{BlockDim.x} = 4$
  - $\text{BlockDim.y} = 3$



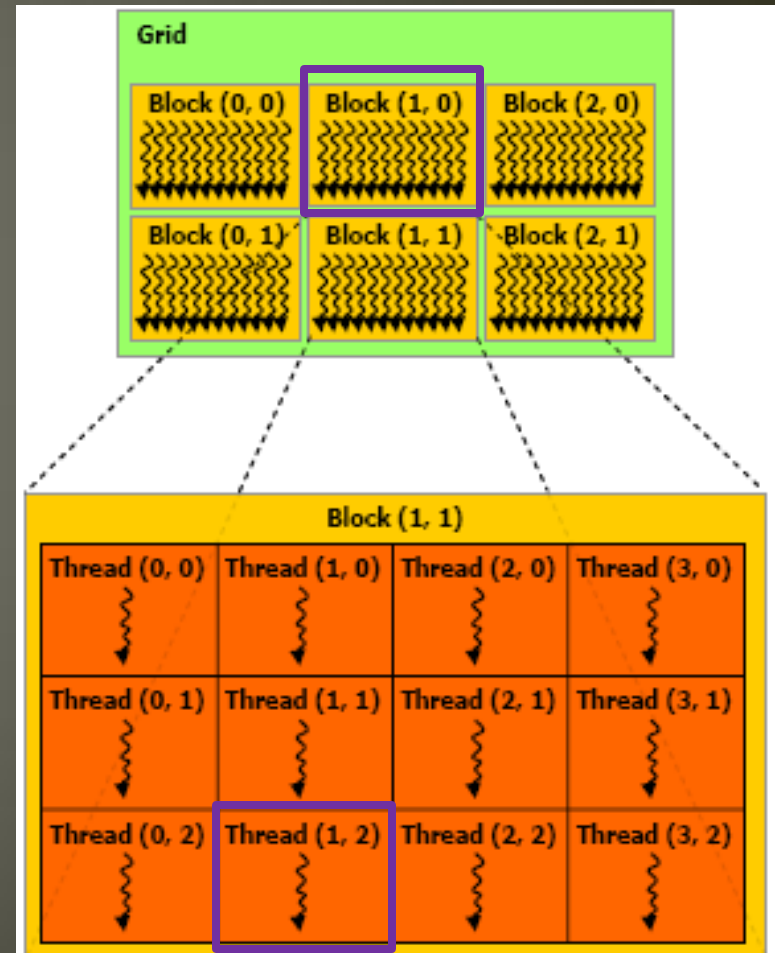
# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$
- $\text{Block\_glob\_id} = \text{BlockIdx.y} \times \text{GridDim.x} + \text{BlockIdx.x} = 0 \times 3 + 1 = 1$ 
  - $\text{BlockDim.x} = 4$
  - $\text{BlockDim.y} = 3$
- $\text{Nb\_threads\_per\_block} = \text{BlockDim.x} \times \text{BlockDim.y} = 4 \times 3 = 12$



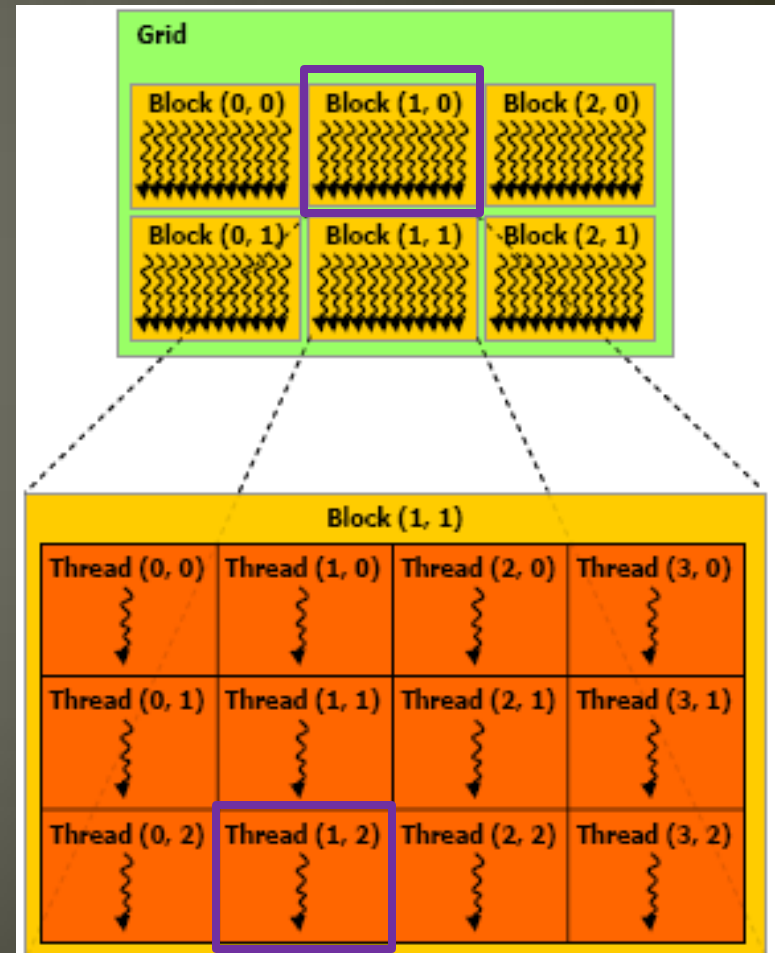
# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$
- $\text{Block\_glob\_id} = \text{BlockIdx.y} \times \text{GridDim.x} + \text{BlockIdx.x} = 0 \times 3 + 1 = 1$ 
  - $\text{BlockDim.x} = 4$
  - $\text{BlockDim.y} = 3$
- $\text{Nb\_threads\_per\_block} = \text{BlockDim.x} \times \text{BlockDim.y} = 4 \times 3 = 12$ 
  - $\text{threadIdx.x} = 1$
  - $\text{threadIdx.y} = 2$
  - $\text{BlockDim.x} = 4$



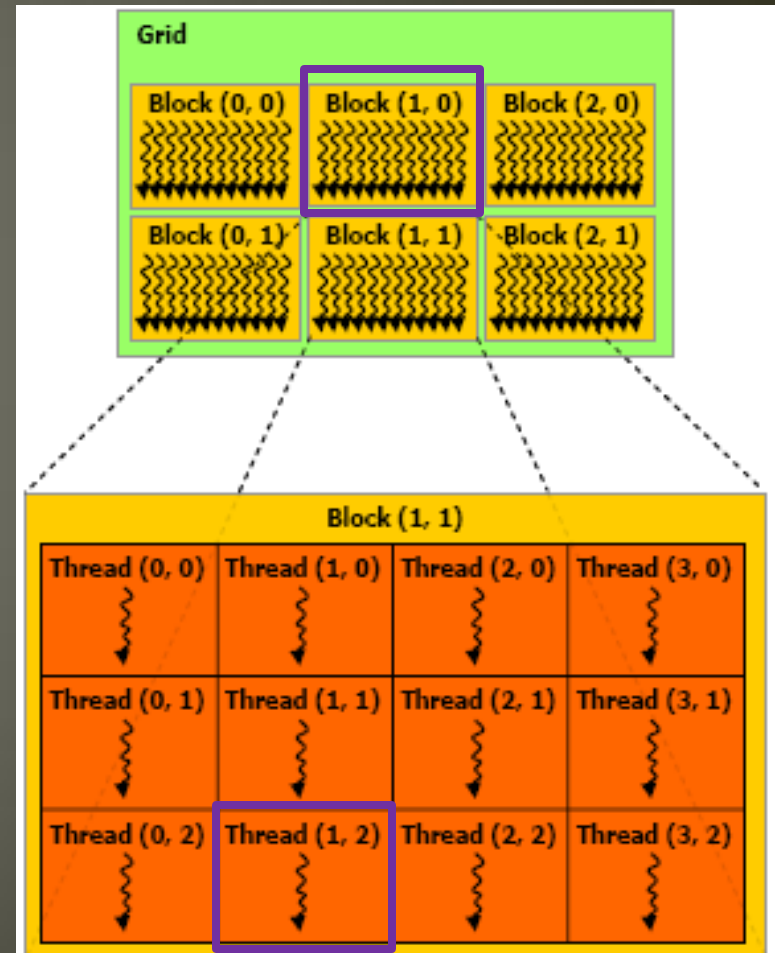
# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$
- $\text{Block\_glob\_id} = \text{BlockIdx.y} \times \text{GridDim.x} + \text{BlockIdx.x} = 0 \times 3 + 1 = 1$ 
  - $\text{BlockDim.x} = 4$
  - $\text{BlockDim.y} = 3$
- $\text{Nb\_threads\_per\_block} = \text{BlockDim.x} \times \text{BlockDim.y} = 4 \times 3 = 12$ 
  - $\text{threadIdx.x} = 1$
  - $\text{threadIdx.y} = 2$
  - $\text{BlockDim.x} = 4$
- $\text{Thread\_loc\_id} = \text{threadIdx.y} \times \text{BlockDim.x} + \text{threadIdx.x} = 2 \times 4 + 1 = 9$



# Hiérarchie de threads

- Calculons l'ID global du thread encadré
  - $\text{BlockIdx.x} = 1$
  - $\text{BlockIdx.y} = 0$
  - $\text{GridDim.x} = 3$
  - $\text{GridDim.y} = 2$
- $\text{Block\_glob\_id} = \text{BlockIdx.y} \times \text{GridDim.x} + \text{BlockIdx.x} = 0 \times 3 + 1 = 1$ 
  - $\text{BlockDim.x} = 4$
  - $\text{BlockDim.y} = 3$
- $\text{Nb\_threads\_per\_block} = \text{BlockDim.x} \times \text{BlockDim.y} = 4 \times 3 = 12$ 
  - $\text{threadIdx.x} = 1$
  - $\text{threadIdx.y} = 2$
  - $\text{BlockDim.x} = 4$
- $\text{Thread\_loc\_id} = \text{threadIdx.y} \times \text{BlockDim.x} + \text{threadIdx.x} = 2 \times 4 + 1 = 9$
- $\text{Thread\_glob\_id} = \text{Block\_glob\_id} \times \text{Nb\_threads\_per\_block} + \text{Thread\_loc\_id} = 1 \times 12 + 9 = 21$





# 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 N) {  
  
    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 ?

# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```

# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```

# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```

# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```

# Programme CUDA complet

## HOST

```
void vectAdd(
    double * a, double * b, double * c, int N) {

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

    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);

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

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

## DEVICE

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

    int i ;

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

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

# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```



# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```

# Programme CUDA complet

## HOST

```
void vectAdd(  
    double * a, double * b, double * c, int N) {  
  
    double * d_a ;  
    double * d_b ;  
    double * d_c ;  
  
    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);  
  
    cudaMemcpy(c, d_c, N*sizeof(double),  
        cudaMemcpyDeviceToHost);  
  
    cudaFree(d_a);  
    cudaFree(d_b);  
    cudaFree(d_c);  
}
```

## DEVICE

```
__global__ void vecAddKernel( double  
    *a, double *b, double *c, int N) {  
  
    int i ;  
  
    i = blockIdx.x * blockDim.x +  
        threadIdx.x ;  
  
    if ( i<N ) {  
        c[i] = a[i]+b[i];  
    }  
}
```

# Programme CUDA complet

## HOST

```
void vectAdd(
    double * a, double * b, double * c, int N) {

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

    cudaMalloc((void **)&d_a, 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<<<32,64>>>(d_a, d_b, d_c, N);

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

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
}
```

## DEVICE

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

    int i ;

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

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