Programmation GPU et traitement d'images

Cours de Master II

Intelligence Artificielle et Multimédia (IAM)

Par : Pr. Rachid SEGHIR

Département d'informatique, Université de Batna 2

r.seghir@univ-batna2.dz

Objectif et Contenu du cours

Objectif du cours

A l'issue de ce cours, vous aurez acquis des connaissances de bases sur les GPU du point de vue architecture, programmation et applications. L'accent sera mis sur l'initiation à la matière et non pas sur son aspect avancé.

Plan du cours

- Introduction et motivation
- Architecture des GPU
- Programmation des GPU
  - Introduction à CUDA C
  - Coopération entre threads
  - Mémoire de données constante et Mémoire de texture
- Application au traitement d'images

Références

- CUDA par l' exemple : une introduction à la programmation parallèle de GPU (Jason Sanders, Edward Kandrot, 2011)
- NVIDIA CUDA C Programming Guide, Version 9.0 September 2017

Introduction et Motivation

L’ère du traitement parallèle

- Le monde informatique a quasiment basculé vers le traitement parallèle depuis l’introduction des premiers processeurs multicœurs aux alentours de l’année 2005.

- Le basculement vers la technologie multicœurs a principalement été motivé par l’impossibilité de continuer d’augmenter la fréquence de l’horloge des processeurs (mono-cœurs) sans se débarrasser du problème de la dissipation thermique,

- Aujourd’hui même les netbooks, tablettes, et smartphones d’entrée de gamme sont dotés de processeurs multicœurs.

- Le traitement parallèle n’est donc plus la spécificité des supercalculateurs, c’est l’ère du calcul parallèle

L’avènement des GPUs (Graphics Processing Units)

- Parallèlement au développent vécu par la technologie des CPUs (Central Processing Units), les ordinateurs personnels ont commencé à se doter d’accélérateurs graphiques 2D afin d’améliorer l’affichage et la fluidité des systèmes d’exploitation graphiques au début des années 1990.

- A la même époque, la société Silicon Graphics popularisa l’utilisation des graphismes 3D dans le monde professionnel (applications gouvernementales, militaires, scientifiques, médicales, etc.). En 1992 l’interface de programmation OpenGL a été introduite par Silicon Graphics afin de faciliter la programmation de son matériel.

- Au milieu des années 1990 (avec l’avènement de SE graphiques comme Windows 95 et la baisse des prix des PC), la demande de graphisme en 3D s’est accrue dans le monde de l’informatique personnelle (jeux vidéos).

L’avènement des GPUs (Graphics Processing Units) (2)

- La popularité du graphisme 3D au milieu des années 90 a poussé des sociétés comme NVIDIA, ATI Technology et 3dfx Interactive à commencer de produire des accélérateurs graphiques à des prix suffisamment abordables pour intéresser le grand publique.

- La sortie de la carte GeForce 256 de NVIDIA repoussa encore plus loin les limites des processeurs graphiques, où un processeur graphique pouvait –pour la première fois- exécuter directement les traitements concernant les transformations et la lumière.

- Mais la sortie de la série GeForce 3 de NVIDIA en 2001 est considérée comme l’étape la plus importante dans l’évolution de la technologie GPU. Pour la première fois, les développeurs pouvaient contrôler les traitement effectués sur leur GPU (Standard DirectX 8.0 de Microsoft).

Traitements généraux sur GPU

- L'apparition de GPU permettant de programmer les traitements déclencha de nombreuses recherches sur la possibilité d'utiliser les circuits graphiques pour autre chose qu'un affichage graphique OpenGL ou DirectX.

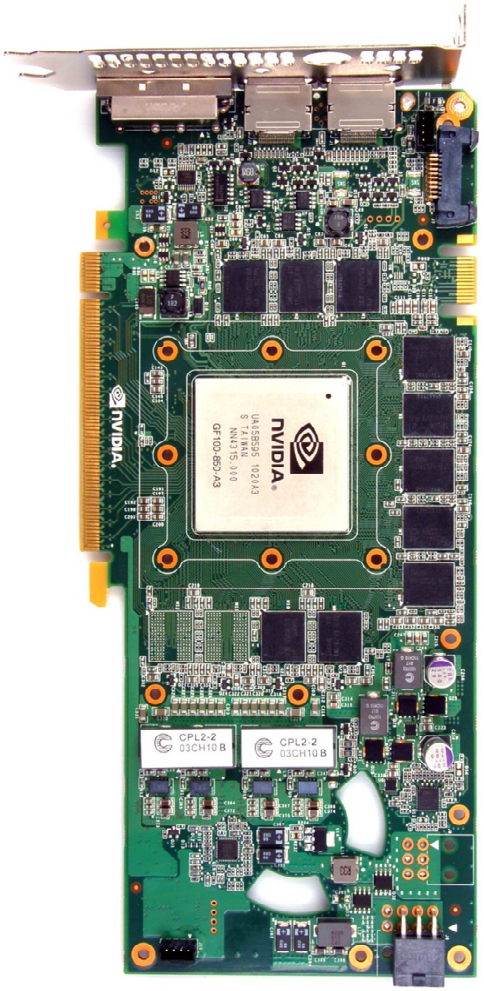
- L'idée c'est de pouvoir utiliser le GPU pour exécuter des traitements généraux, où ce qui est connu sous le nom de GPGPU (General-Purpose Computing on Graphics Processing Units).

- Ce n'est que cinq ans après la sortie de GeForce 3 de NVIDIA (2006) que le traitement sur GPU allait devenir accessible à tous.

- En novembre 2006, NVIDIA révéla le premier GPU (GeForce 8800 GTX) reposant sur l'architecture CUDA (Compute Unified Device Archetecture), et dédié aux traitements généraux. Les langages associés sont CUDA C/C++ et CUDA Fortran.

Exemple de GPU pour traitements Généraux

- NVIDIA Tesla C2070 (pour Calcul Scientifique)



Caractéristiques techniques: NVIDIA Tesla C2070/50

Facteur de forme	Facteur de forme 9,75" PCIe x16
Nbre de GPU Tesla	1
Nombre de cœurs CUDA	448
Fréquence des cœurs CUDA	1.15GHz
Performances de la virgule flottante en double précision	515 Gigaflops
Performances (en pic) de la virgule flottante en simple précision	1.03 TFLOPS
Mémoire dédiée totale Tesla C2050 Tesla C2070	 3 Go GDDR5 6 Go GDDR5
Vitesse de la mémoire	1.5 GHz
Interface mémoire	384-bit
Bande passante mémoire	144 Go/sec
Consommation maximale Tesla C2050	 238W
Interface système	PCIe x16 Gen2
Solution thermique	Ventilateur actif
Affichage Dual-Link DVI-I Résolution maximale à 60 Hz	 1 2560x1600
Outils de développement logiciel	Kits d'outils CUDA C/C++/Fortran, OpenCL, DirectCompute, NVIDIA Parallel Nsight™ pour Visual Studio

Exemples de traitements généraux avec CUDA

- Imagerie médicale : La société TechniScan a développé une méthode d'imagerie par ultrasons prometteuse pour un diagnostique très précis. Mais cette solution n'a pas été mise en pratique à cause de la puissance de traitement nécessaire (Ultrasons ->Img 3D)

- Les rêves des fondateurs de TechniScan on été transformés en réalité grâce au système Svara qui utilise deux processeurs Tesla C1060 permettant de traiter 35 Go de données d'un scan de 15 mn pour produire, dans seulement 20 mn, une image 3D très détaillée.

- Dynamique des fluides : La conception de rotors et de pales efficace a resté pendant longtemps réservée aux seuls chercheurs ayant accès aux grands (et coûteux) Supercalculateurs en raison de la complexité de la simulation du mouvement de l'aire et des fluides autour de ces composants

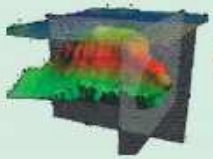
Exemples de traitements généraux avec CUDA (2)

- Graham Pullan et Tobias Brandvik de l'université de Cambridge ont remarqué que l'utilisation de l'architecture CUDA sur des grappes de GPU permettait d'accélérer le traitement de la dynamique des fluides, et d'atteindre des gains en performances impressionnants, à des coûts beaucoup plus réduits.

- Protection de l'environnement : L'université Temple de Philadelphie a travaillé avec les sociétés Procter et Gamble pour simuler les interactions des molécules tensioactives (composants essentiels des produits de nettoyage ayant un effet désastreux sur l'environnement).

- Le logiciel de simulation HOOMD sur deux GPU Tesla de NVIDIA a permis d'aboutir à des performances équivalentes à celle de cluster BlueGene/L d'IBM (1024 CPU). Les années à venir devrait voir apparaitre des produits nettoyants dont l'efficacité est accrue tout en ayant un impact moindre sur l'environnement.

Domaines d’application des GPGPU



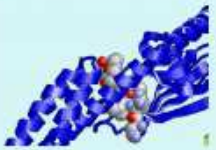
Computational Geoscience

Computational Chemistry

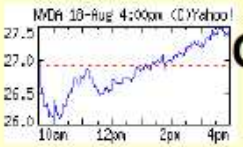
Computational Medicine

Computational Modeling

Computational Physics



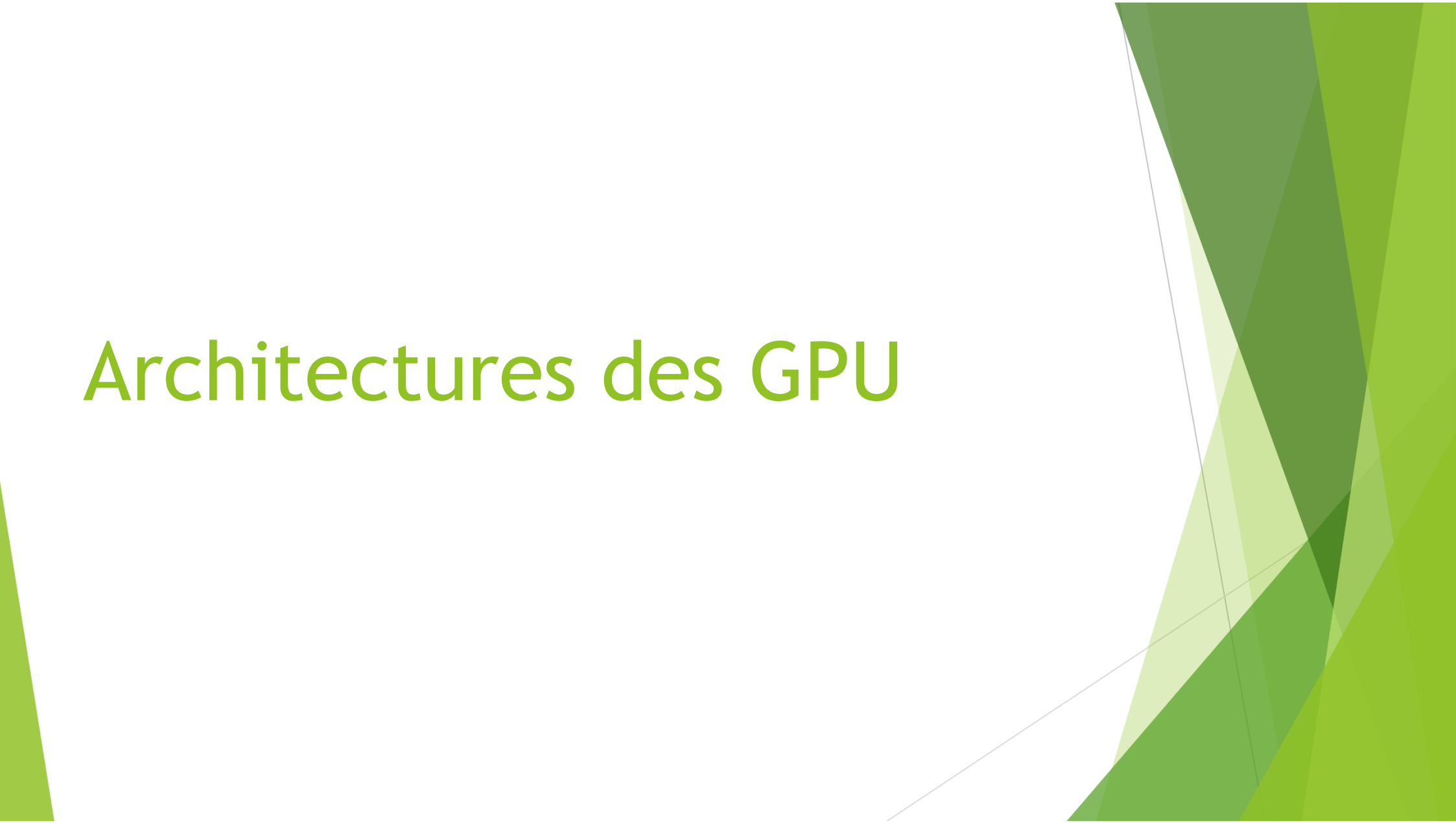
Computational Biology



Computational Finance

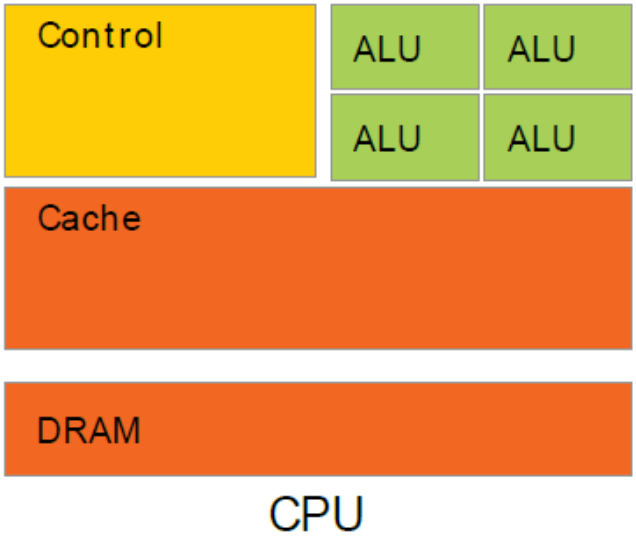
Image Processing

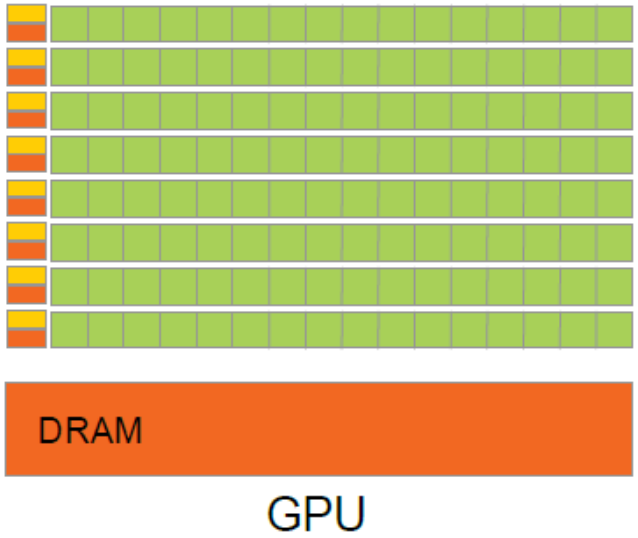
Architectures des GPU



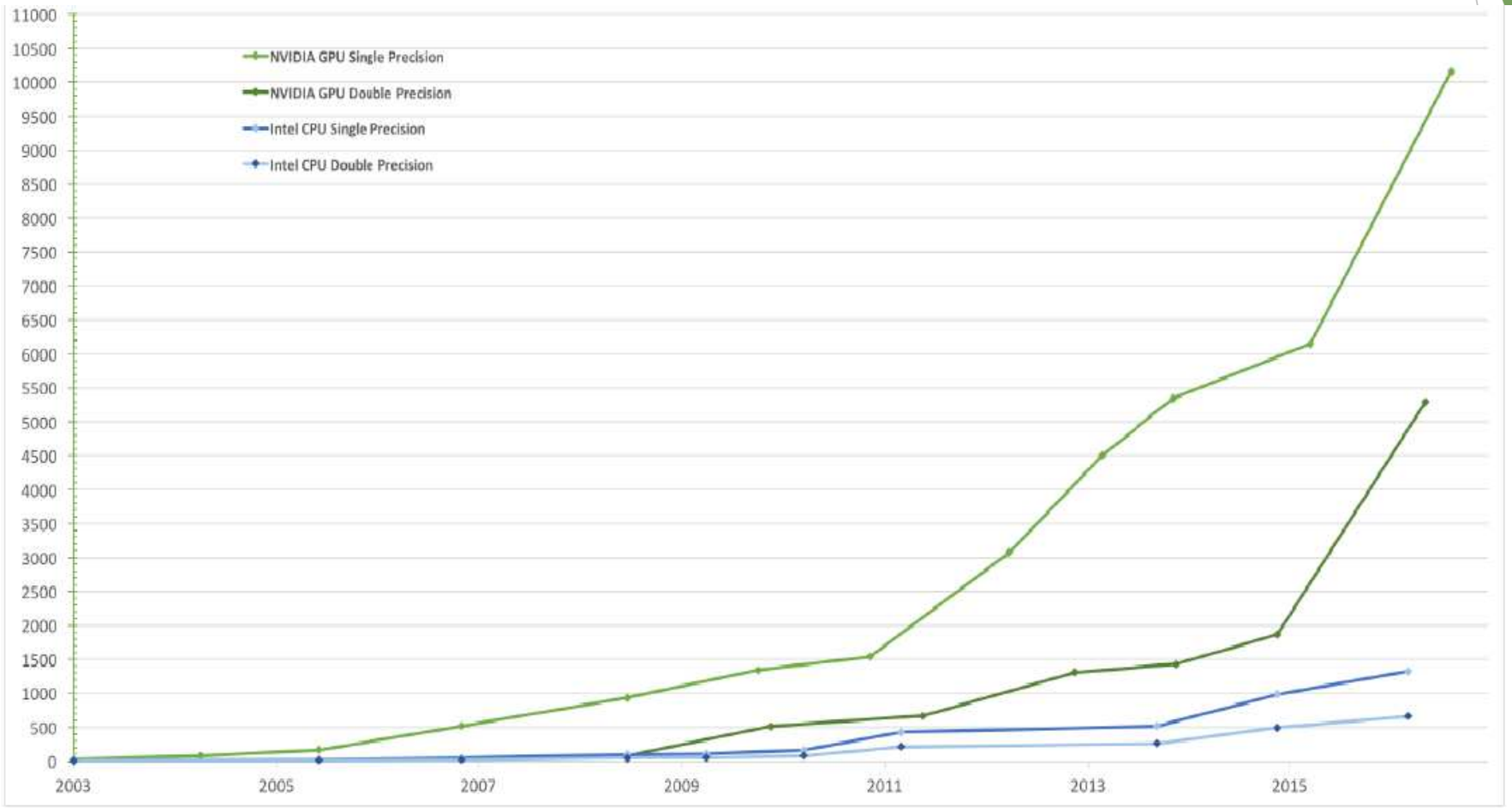
Architecture GPU vs Architecture CPU

- Les processeurs graphiques, spécialisés dans le calcul intensif, dédient plus de transistors au traitements de données que les processeurs classiques,
- Moins de transistors sont dédiés au cache de données et au control de flot
- Ce qui explique le gain significatif en performances (puissance de calcul et bande passante de la mémoire)

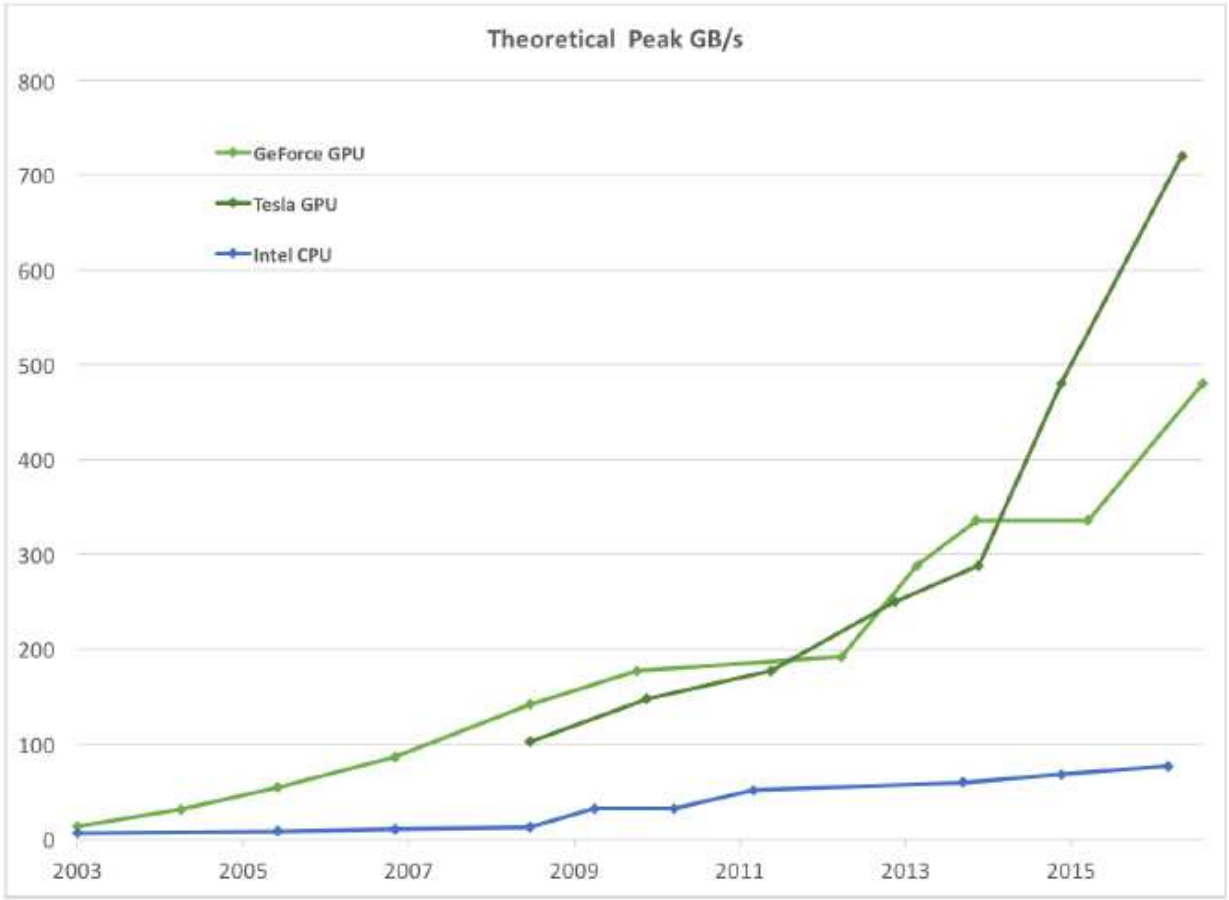




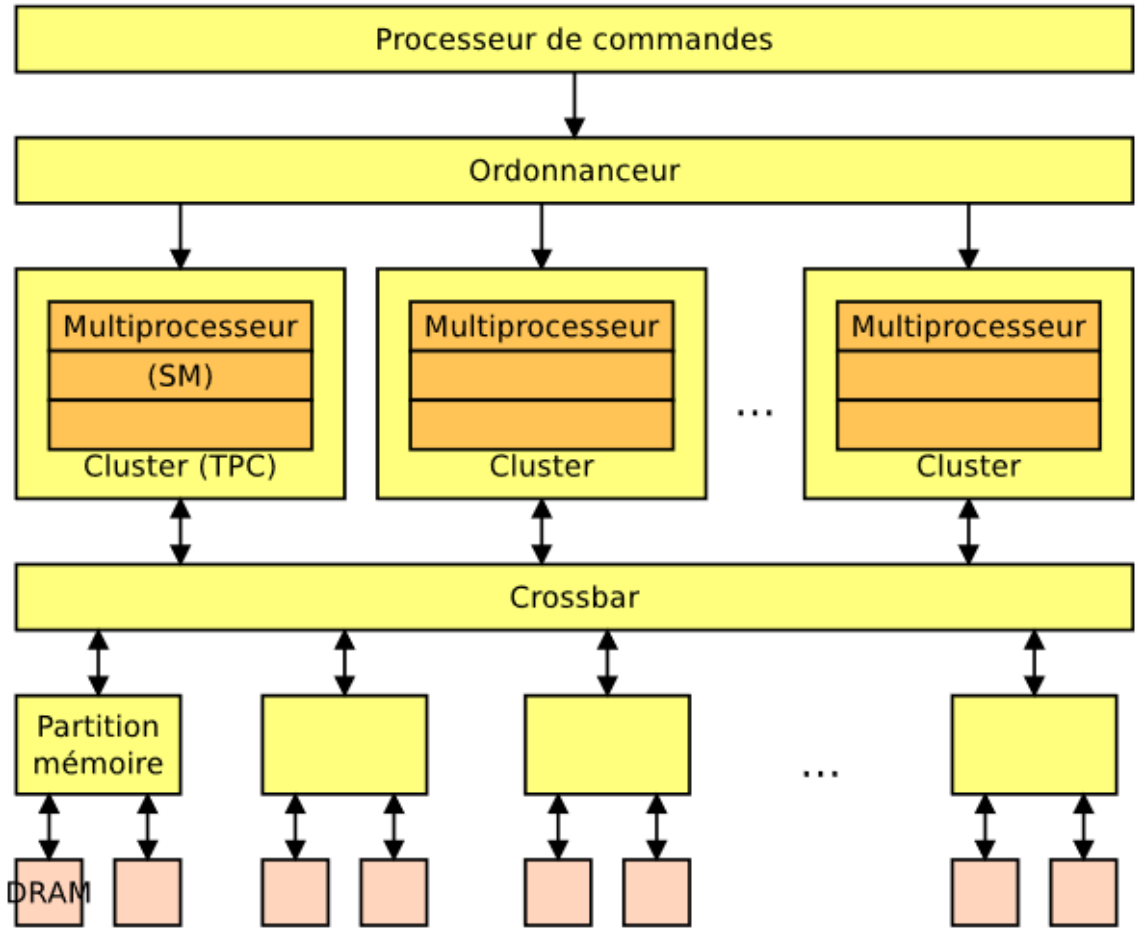
Nombre de GFLOPS/s pour le CPU et le GPU



La Bande passante de la mémoire pour le CPU et le GPU



Vue générale de l'architecture d'un GPU (Tesla)



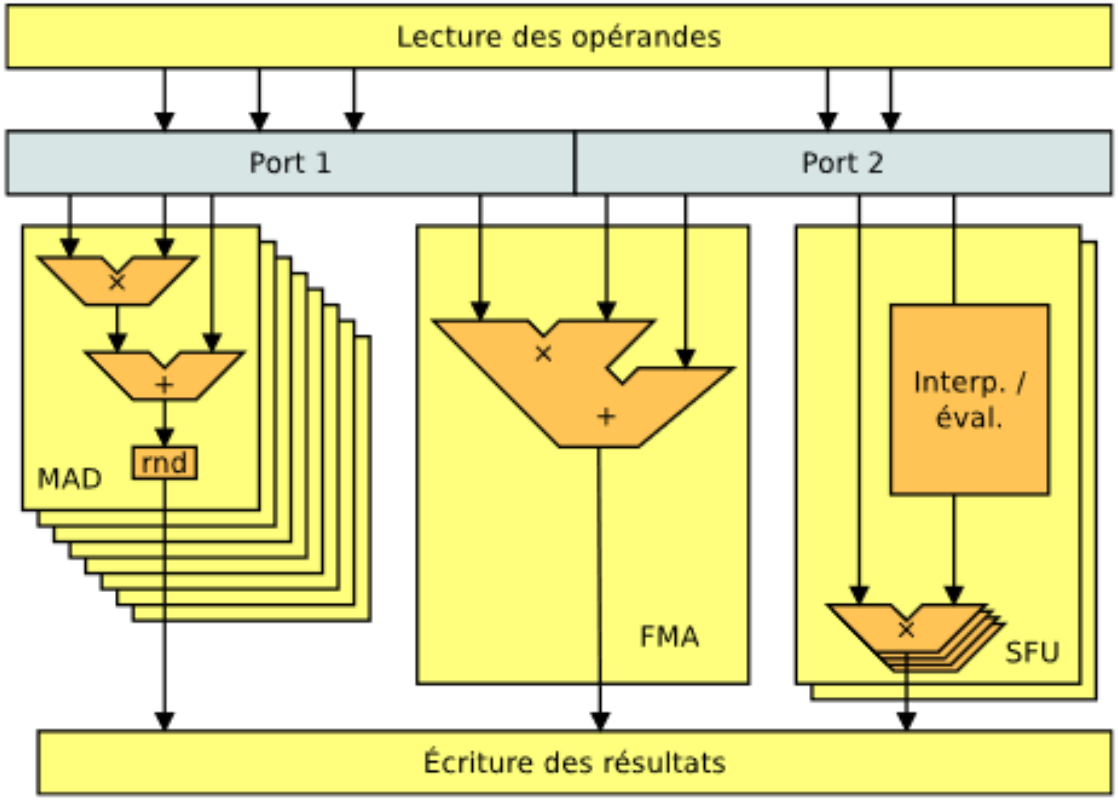
Architecture d'un GPU (Partie Calcul)

- La partie calcul d'un GPU est composée d'une hierarchie de cœurs
- Au niveau le plus elevé, le GPU contient jusqu'à 10 TPC (Texture/Processor Cluster)
- Les differents TPC sont connectés aux autres composants (mémoires) par réseau d'interconnexion en croix
- Chaque TPC contient une unité d'accès à la memoire, disposant de son propre cache de premier niveau et de ses unités de calcul d'adresse et filtrage de texture, et plusieurs cœurs de calcul ou SM (Streaming Multiprocessors) ou encore Multiprocesseurs
- Chaque Multiprocesseur est un processeur autonome SIMD multi-thread (contenant plusieurs coeurs)

Architecture d'un GPU (Unités d'exécution)

- Des unités MAD (Matiplication/ADdition) sont chargées des calculs arithmétiques généralistes
- Il s'agit d'un pipeline construit autour d'un multiplieur 24×24 suivi d'un additionneur puis d'une unité d'arrondi.
- L'unité MAD est capable d'effectuer avec une latence et un débit constants aussi bien les instructions entières (arithmétique, logique, décalages) que virgule-flottante (multiplication-addition, min, max, conversions) en simple précision
- On distingue aussi l'unité (multiplication-addition) en double précision dite FMA, et l'unité d'évaluation de fonctions élémentaires SFU

Schéma de l’unités d’exécution)



Architecture d'un GPU (Interface)

- Le GPU Tesla étant un co-processeur spécialisé, n'est pas capable de fonctionner d'une manière autonome, il est dirigé par un pilote s'exécutant sur le ou les CPU
- Le pilote contrôle l'état (registres de configuration) du GPU, lui envoie des commandes, et peut recevoir des interruptions indiquant la complétion d'une tâche
- le GPU est controlé au travers d'une file de commandes selon un schéma producteur-consommateur classique. La file est typiquement placée en memoire centrale
- Le pilote graphique ou GPGPU se charge d'ajouter des commandes dans la file, tandis que le processeur de commandes du GPU retire les commandes à l'autre extrimété de la file
- Les commandes peuvent être des commandes de configuration, des copies mémoires, des lancements de kernels (noyau), etc,

Architecture d'un GPU (Ordonnanceur)

- Un ordonnanceur est une unité du GPU chargée de répartir les blocs (groupes de threads) à exécuter sur les multiprocesseurs

- Avant d'envoyer un signal de début d'exécution aux multiprocesseurs, l'ordonnanceur procède à l'initialisation des registres et des mémoires partagées.
  - Initialisation des arguments du Kernel
  - Initialisation des coordonnées du bloc
  - Initialisation de la taille des blocs de la gille
  - Initialisation des coordonnées du thread à l'intérieur du bloc

- La politique d'ordonnancement des blocs est de type round-robin, et l'ordonnanceur global attend la terminaison de tous les blocs en cours d'exécution sur le GPU avant d'ordonnancer le groupe de blocs suivant

Architecture d'un GPU (Hiérarchie mémoire)

- Dans un processeur gaphique, on trouve différents types de mémoires :
- Mémoire globale : est la mémoire DRAM accéssible par tous les Multiprocesseurs du GPU
- Mémoire partagée : est la mémoire partagée par tous les threads appartenant à un Multiprocesseur
- Mémoire locale (privée) : c'est une mémoire propre au thread qui ne peut être accédée que par ce dernier
- Mémoire de données constantes : sert à stocker les données qui ne seront pas modifiées au cours de l'exécution d'un noyau (Kernel).
- Mémoire de textures : est une autre variante de mémoire en lecture seule permettant d'améliorer les performances et de déminuer le trafic mémoire de certaines situations particuières
- il y a autant de héirarchies mémoires pour ces différents types de mémoires

Architecture d'un GPU (Hiérarchie mémoire)

Une hiérarchie de mémoires de tailles et temps d'accès très variables

A partir de l'architecture FERMI

Comparé à la vitesse de calcul du GPU

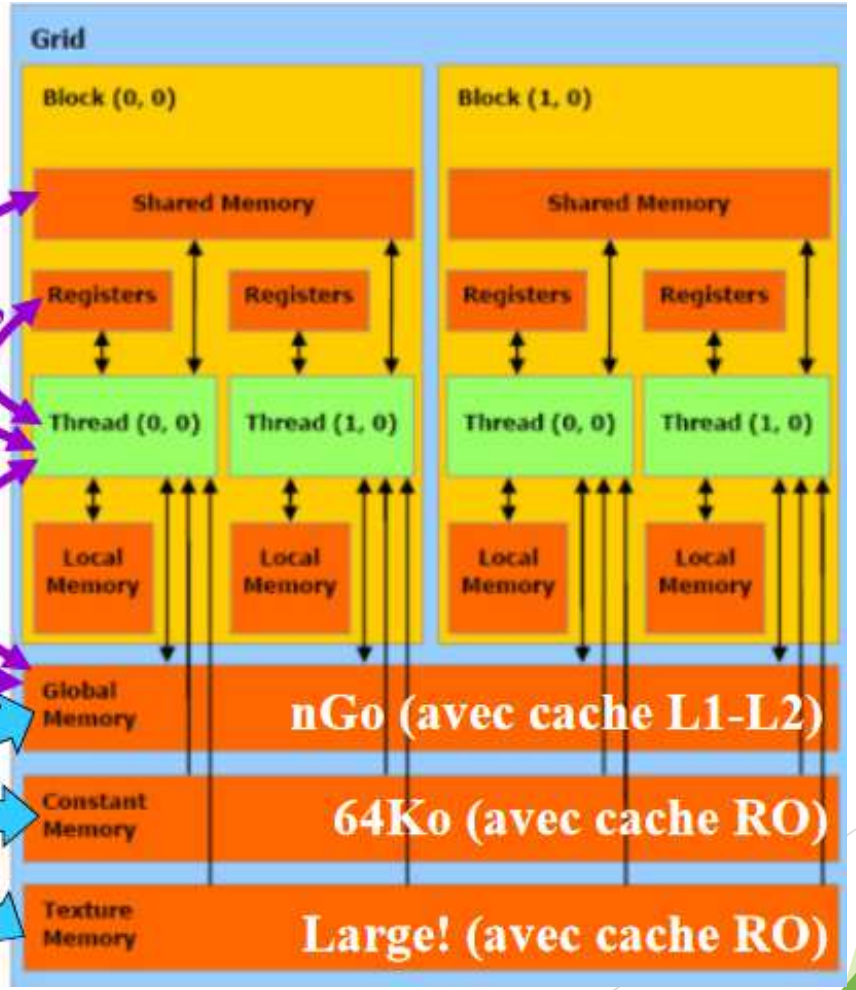
Rapide et souple

'0s'

Moyen-lent & contraint

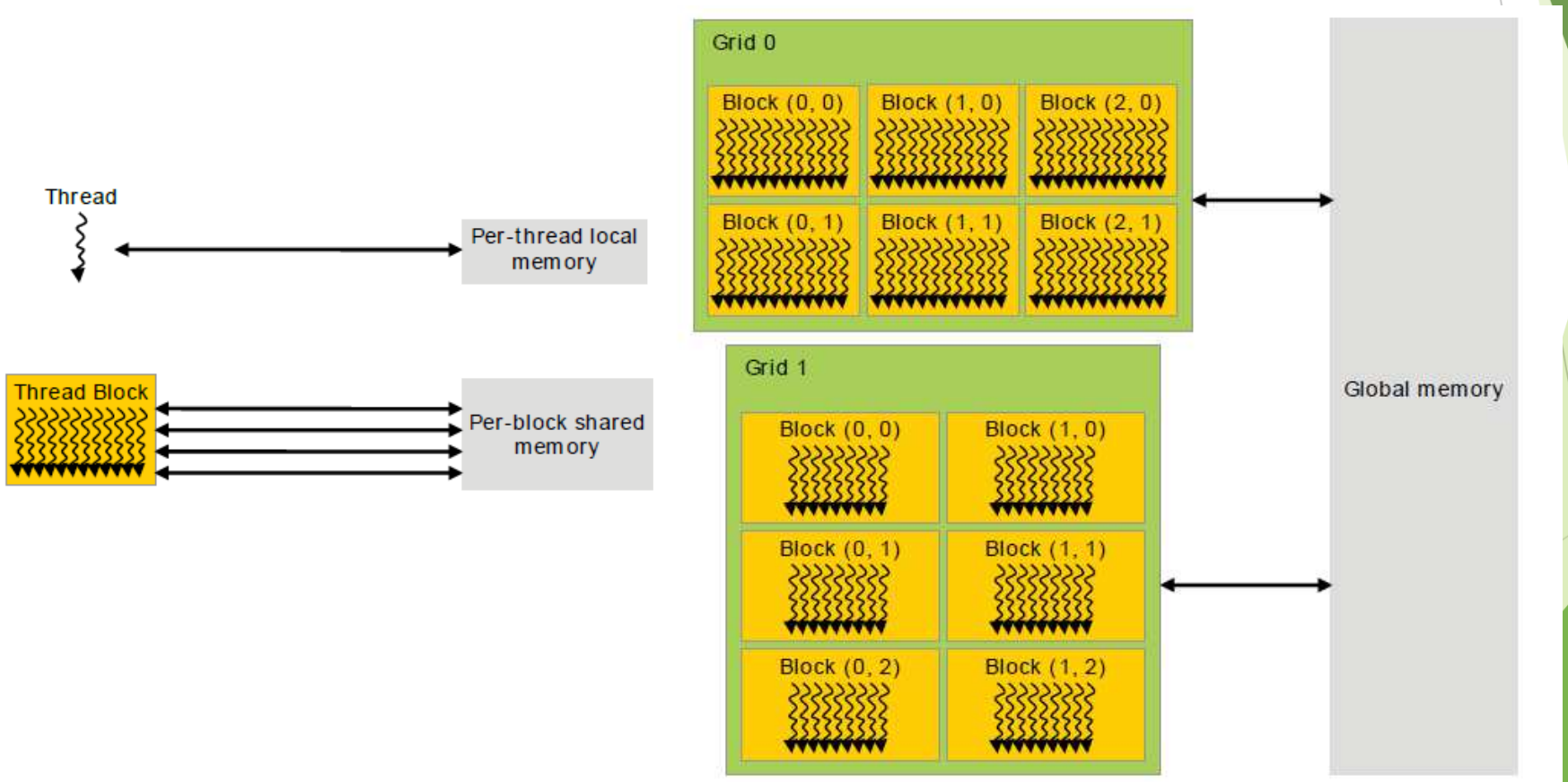
Très lent

CPU + RAM



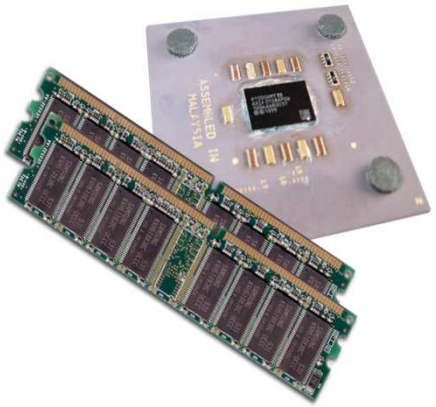
Vue CUDA de l'architecture d'un GPU

- Les threads d'un programme sont réparties sur des grilles contenant des blocs qui eux même contiènnent des threads (les dimensions ne sont pas fixées)



Modèle de Programmation : Calcul hétérogène

- Terminologie :
  - *Host* Le CPU et sa mémoire (host memory)
  - *Device* Le GPU et sa mémoire (device memory)



Host

Device

Modèle de Programmation : Calcul hétérogène (2)

- Le programme est constitué d'une fonction parallèle et d'une succession de codes séquentiels et parallèles s'exécutant sur le Host et le Device (respectivement)

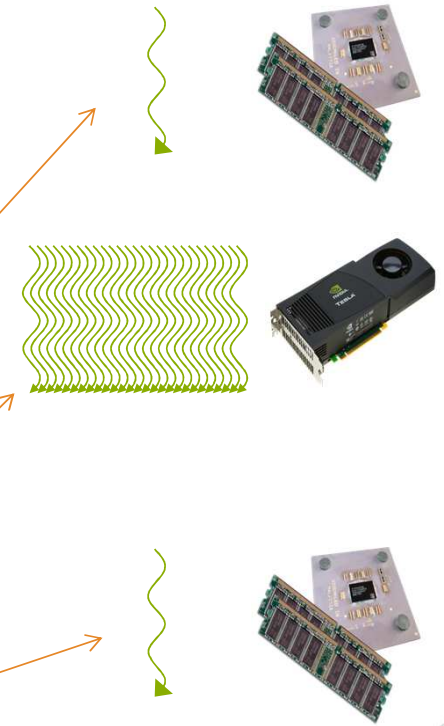
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N       1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x  < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

Fn parallèle

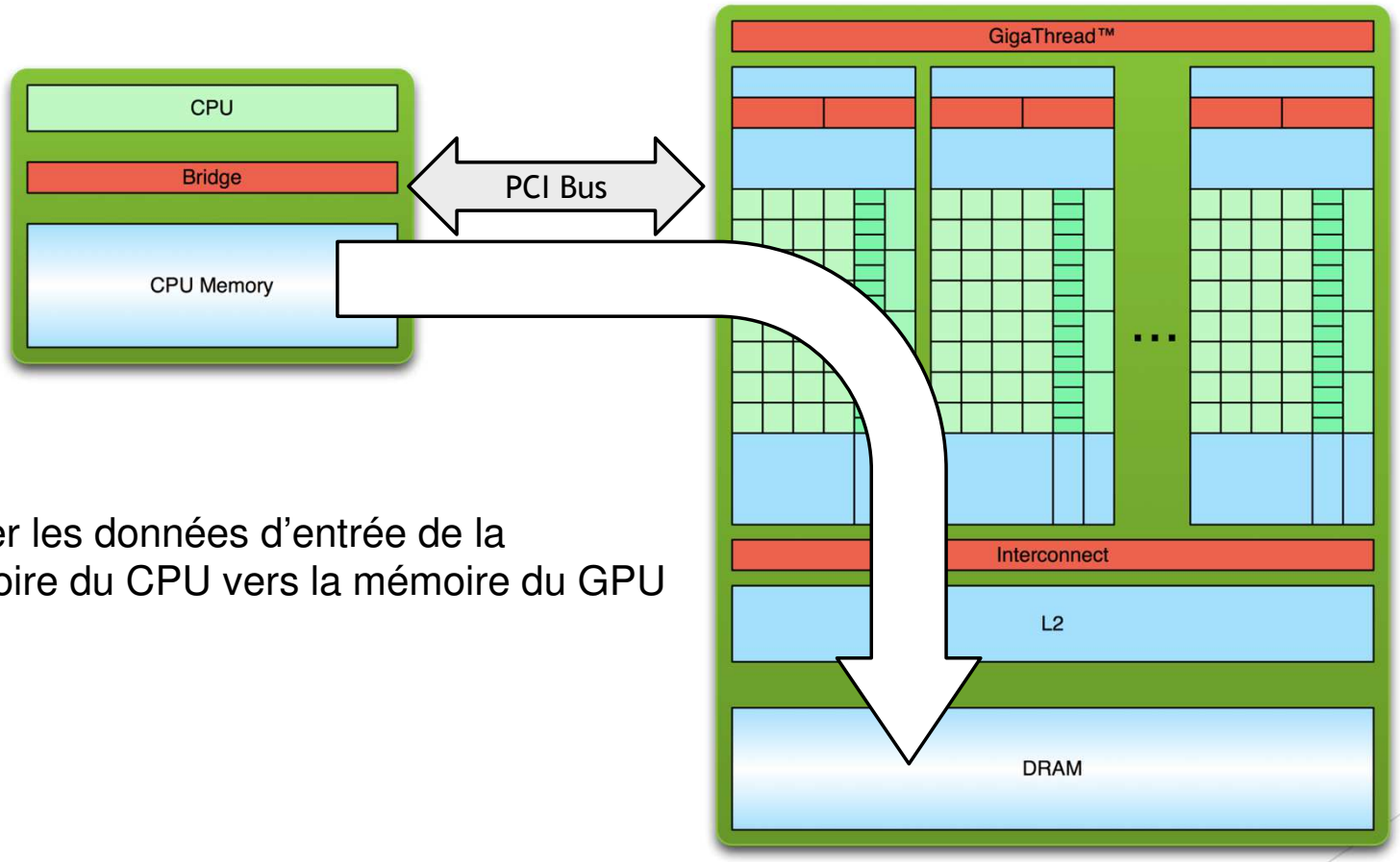
Code séquentiel

Code parallel

Code séquentiel

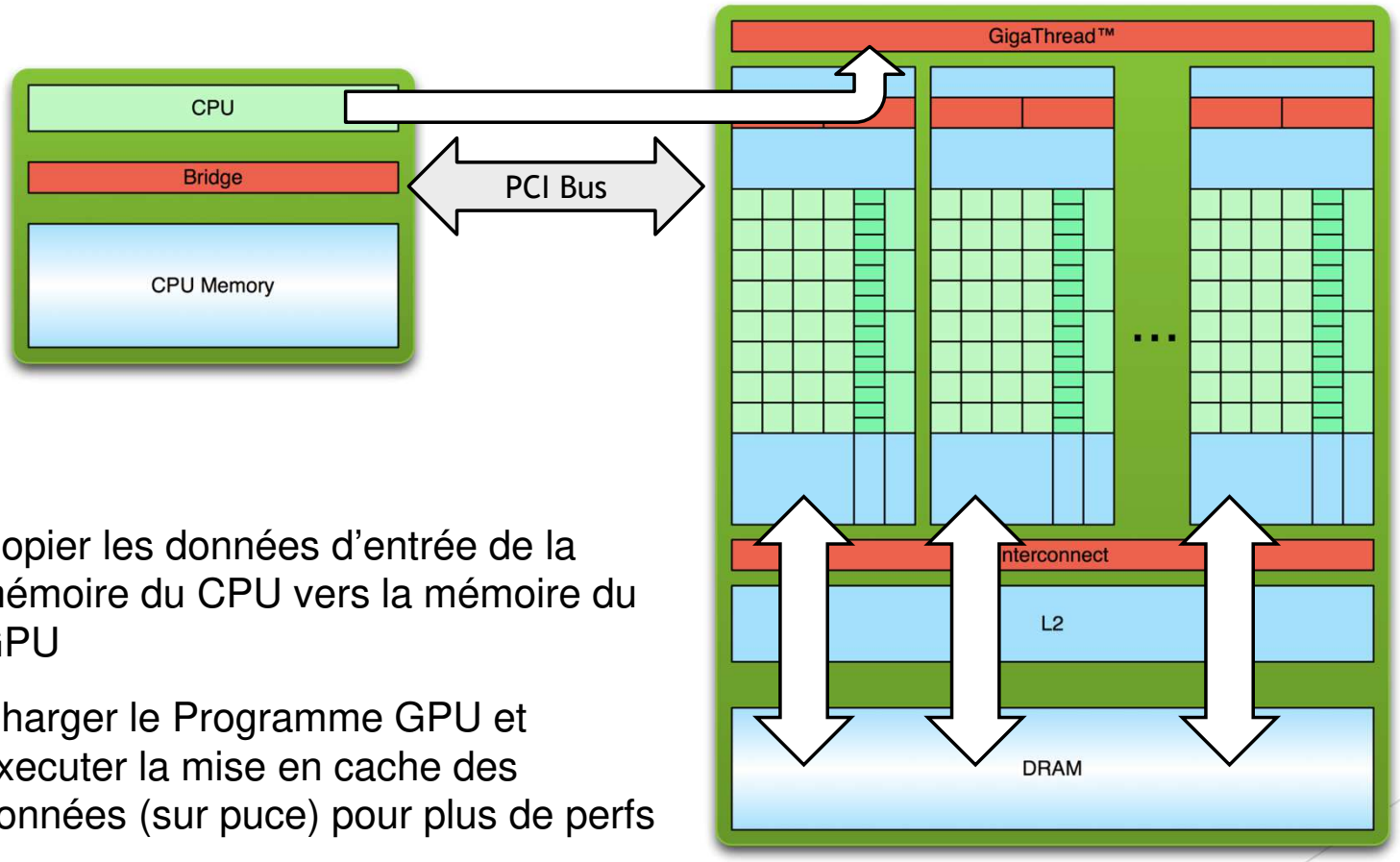


Fonctionnement du calcul hétérogène



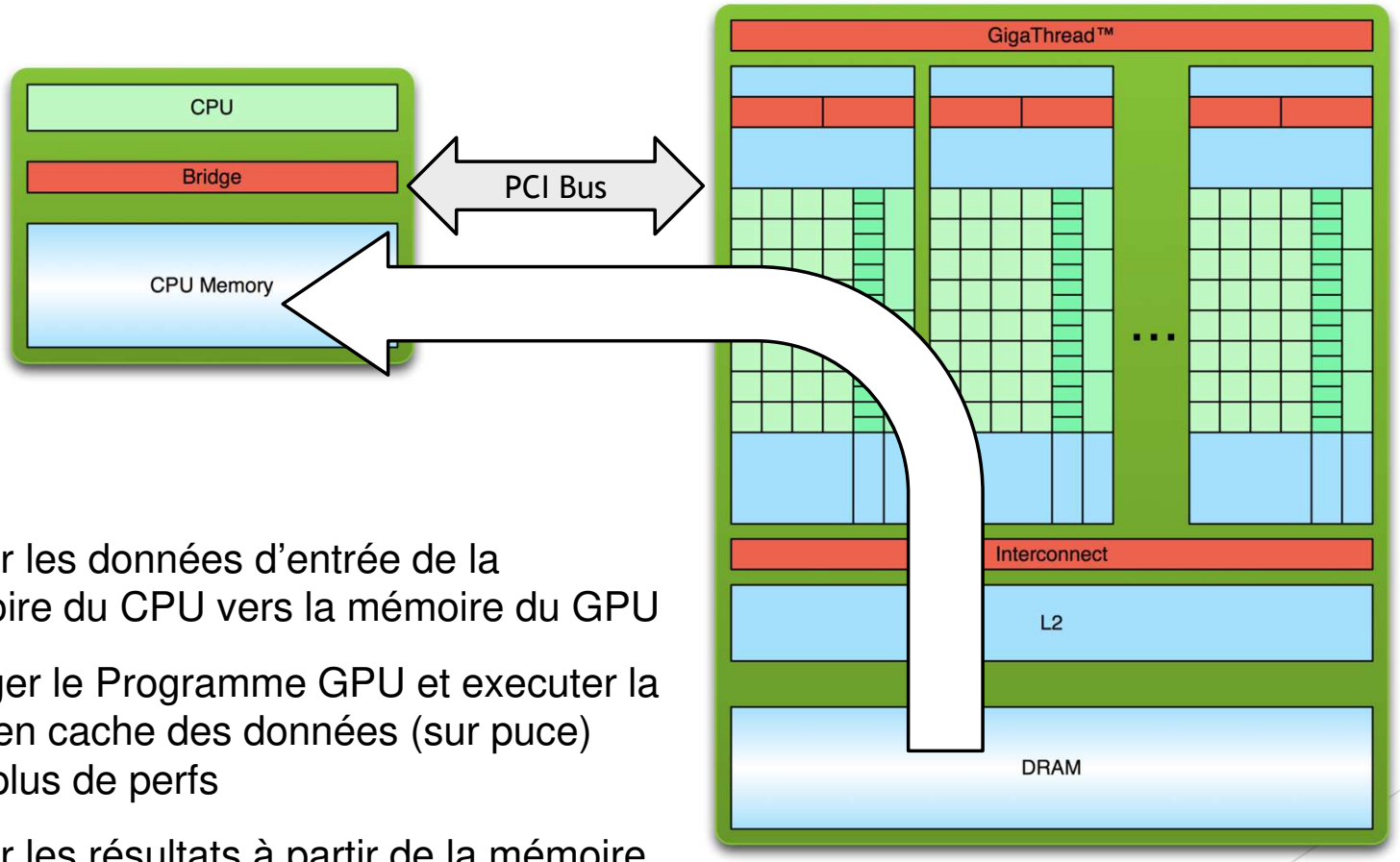
1. Copier les données d'entrée de la mémoire du CPU vers la mémoire du GPU

Fonctionnement du calcul hétérogène (2)



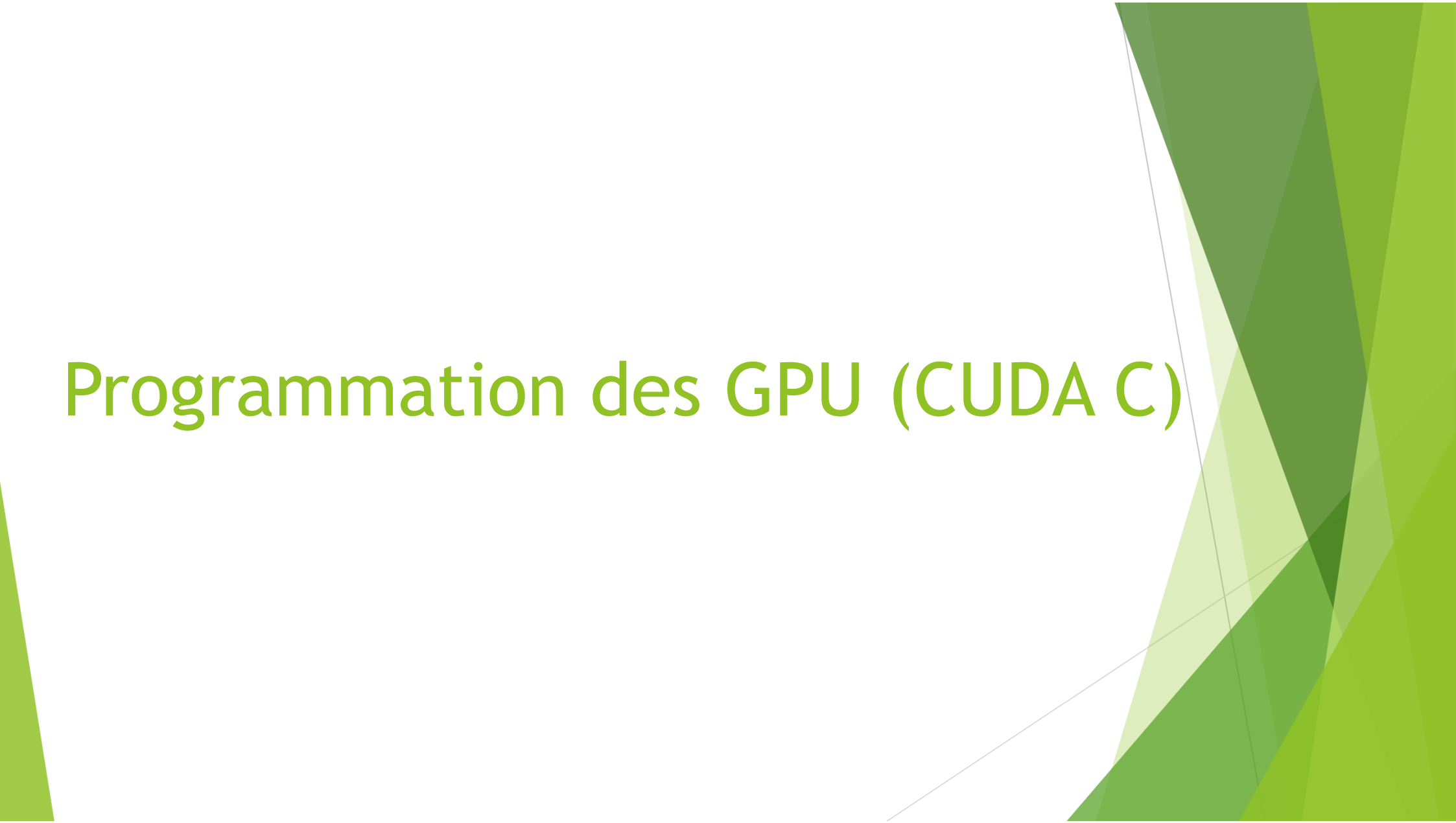
1. Copier les données d'entrée de la mémoire du CPU vers la mémoire du GPU
2. Charger le Programme GPU et executer la mise en cache des données (sur puce) pour plus de perfs

Fonctionnement du calcul hétérogène (2)

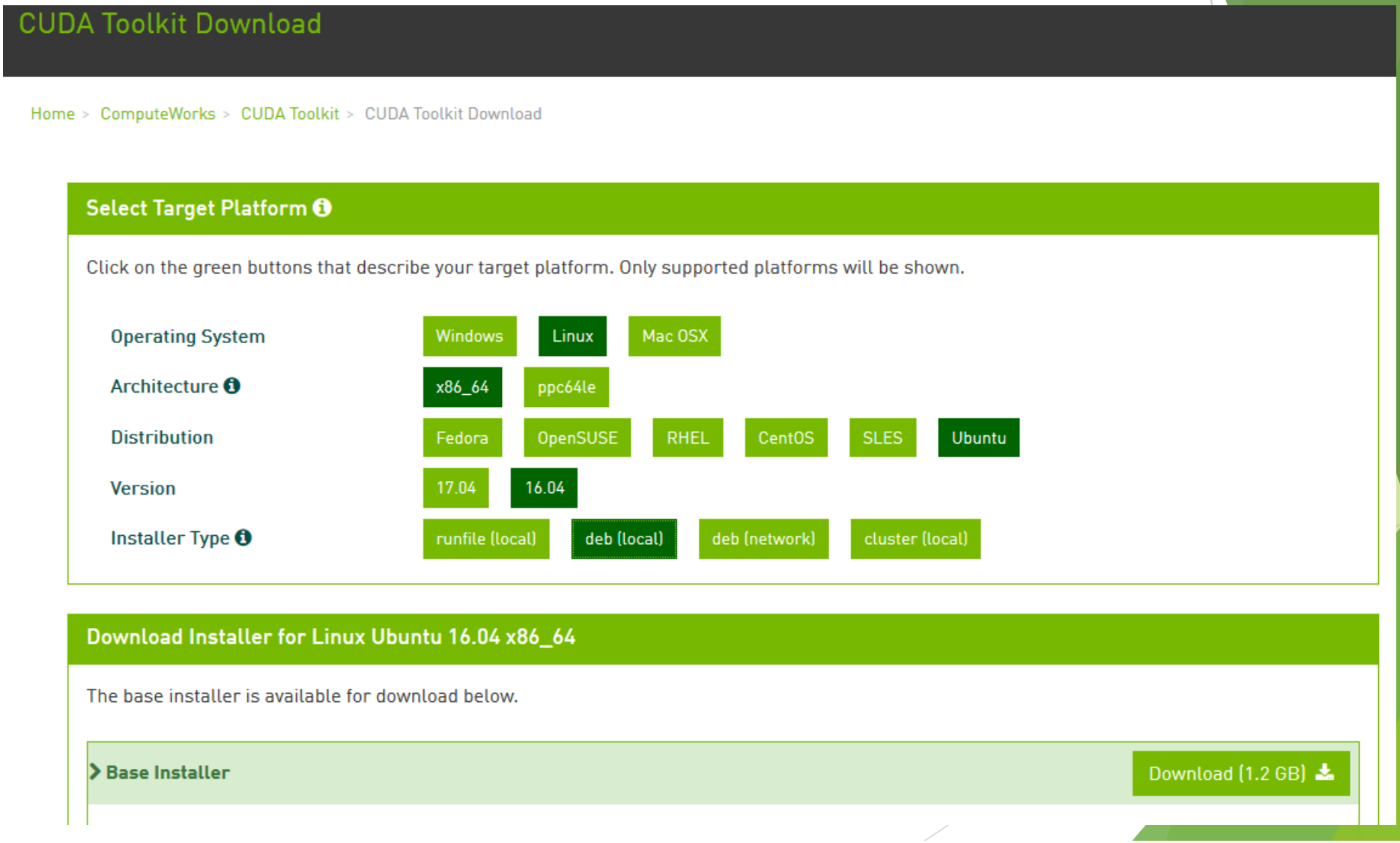


1. Copier les données d'entrée de la mémoire du CPU vers la mémoire du GPU
2. Charger le Programme GPU et executer la mise en cache des données (sur puce) pour plus de perfs
3. Copier les résultats à partir de la mémoire du GPU vers la mémoire du CPU

Programmation des GPU (CUDA C)



Téléchargement de CUDA pour Linux (Ubuntu)



Installation de CUDA (Sous Linux Ubuntu)

- `sudo dpkg -i cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64.deb`
- `sudo apt-key add /var/cuda-repo-9-0-local/7fa2af80.pub`
- `sudo apt-get update`
- `sudo apt-get install cuda`
- `sudo apt-get install nvidia-cuda-toolkit`
- `nvcc  exemple.cu`
- `./a.out`

Affichage des caractéristiques de son GPU

- Avant de commencer à écrire des programmes pour un accélérateur GPU, il est interessant de connaire les différentes caractéristuqes de son dispositif.

- La première chose à connaitre est le nombre de Devices (accélérateurs) compatiubles CUDA qui sont installés sur son système.

- Ceci est possible grace à l'appel de la fonction cudaDeviceCount () comme suit :

```
int count ;
HANDLE_ERROR ( cudaDeviceCount ( & count) ) ;
```

- Ensuite, nous pouvons interroger chaque Device pour récupérer ses caractéristiques à l'aide de l'appel de la fonction : cudaGetDeviceProperties () comme suit :

```
cudaDeviceProp prop ;
HANDLE_ERROR ( cudaGetDeviceProperties ( & prop, i) ) ; // i est le numéro du Device
```

Champs de la structure **cudaDeviceProp**

```
struct cudaDeviceProp    {
    char        name[256];
    size_t      totalGlobalMem;
    size_t      sharedMemPerBlock;
    int         regsPerBlock;
    int         warpSize;
    size_t      memPitch;
    int         maxThreadsPerBlock;
    int         maxThreadsDim[3];
    int         maxGridSize[3];
    size_t      totalConstMem;
    int         major;
    int         minor;
    int         clockRate;
    size_t      textureAlignment;
    int         deviceOverlap;
    int         multiProcessorCount;
    int         kernelExecTimeoutEnabled;
    int         integrated;
    int         canMapHostMemory;
    int         computeMode;
    int         maxTexture1D;
    Int         maxTexture2D[2];
    Int         maxTexture3D[3];
    Int         maxTexture2DArray[3];
    int         concurrentKernels;
}
```

Description des propriétés de l'accélérateur

Propriété de l'accélérateur	Description
int `maxThreadsPerBlock`	The maximum number of threads that a block may contain
int `maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
int `maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
size_t `totalConstMem`	The amount of available constant memory
int `major`	The major revision of the device's compute capability
int `minor`	The minor revision of the device's compute capability
size_t `textureAlignment`	The device's requirement for texture alignment
int `deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
int `multiProcessorCount`	The number of multiprocessors on the device

Description des propriétés de l'accélérateur (2)

Propriété de l'accélérateur	Description
int `kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
int `integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
int `canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space
int `computeMode`	A value representing the device's computing mode: default, exclusive, or prohibited
int `maxTexture1D`	The maximum size supported for 1D textures
int `maxTexture2D[2]`	The maximum dimensions supported for 2D textures
int `maxTexture3D[3]`	The maximum dimensions supported for 3D textures
int `maxTexture2DArray[3]`	The maximum dimensions supported for 2D texture arrays
int `concurrentKernels`	A boolean value representing whether the device supports executing multiple kernels within the same context simultaneously

Premier programme : Hello World

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Il s'agit d'un programme C standard qui s'exécute sur le host (CPU)
- Le Compilateur NVIDIA (nvcc) peut être utilisé pour compiler des programmes sans code pour accélérateur (device)
- Mais ce n'est pas notre objectif !

```
Sortie:

$ nvcc hello_world.cu

$ ./a.out

Hello World!

$
```

Hello World! : avec code pour accélérateur

```
__global__ void mykernel(void) {
}
```

- Le mot clé CUDA C/C++ `__global__` indique une fonction qui :
  - est appelée à partir du code host (qui s'exécute sur le CPU)
  - et qui s'exécute sur le device (GPU)
  - Cette fonction est généralement nommée Noyau (Kernel)

- `nvcc` sépare le code source en composantes hôte (host) et accélérateur (device)
  - Les fonctions Device (e.g. **`mykernel()`**) qui sont traitées par le compilateur NVIDIA (nvcc)
  - Les function Host (e.g. **`main()`**) qui sont traitées par un compilateur standard du Host (e,g, gcc)
- Le compilateur et l'environnement d'exécution CUDA s'occupent de tous les détails de l'appel du code accélérateur à partir de l'hôte

Hello World! : avec code pour accélérateur (2)

```
mykernel<<<1,1>>>();  // appel de la function noyau
```

- L'appel de la fonction kernel est marqué par des chevrons (<<< >>>) et un tuple de nombres : c'est un appel à partir du code hôte vers le code Accélérateur
  - Aussi appelé "kernel launch"
  - Nous allons retourner aux paramètres (1,1) plus tard
- C'est tous ce qu'il faut pour exécuter une fonction sur un GPU

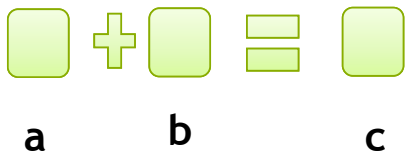
```
__global__ void mykernel(void){
}
int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

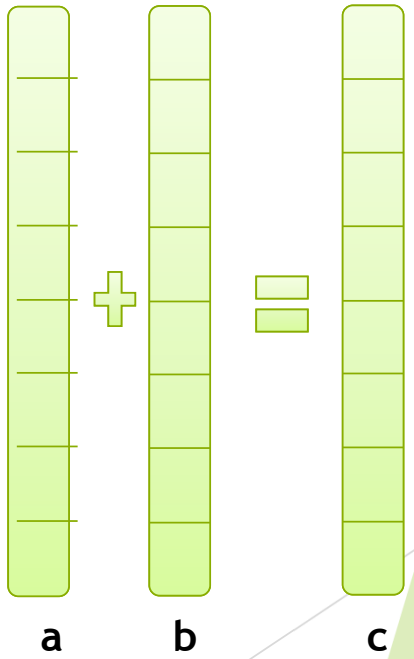
```
Sortie:

$ nvcc hello_world.cu

$ ./a.out

Hello World!

$
```

`mykernel()` ne fait rien, c'est un peu décevant !

Programmation parallèle avec CUDA C/C++

- Le CUDA est bien évidemment conçu pour faire de la programmation massivement parallèle et non pas pour faire des appel à des noyaux qui ne font rien !
- Alors on a besoin d'un exemple plus sophistiqué qui montre comment une fonction noyau peut être utilisée pour réaliser un calcul parallèle
- Nous allons monter cela à travers un exemple de calcul de la somme de deux vecteurs d'entiers
- Nous allons commencer par écrire un programme CUDA C qui calcul la somme de deux entiers par la fonction kernel, avant de l'étendre pour qu'il puisse faire la somme de deux vecteurs

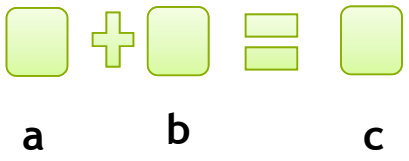




Addition de deux entiers sur l'accélérateur

- Ceci est une simple fonction kernel qui fait l'addition de deux nombres entiers

```
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```



- Comme auparavant, **`__global__`** est un mot clé CUDA C/C++ qui signifie que :
  - **`add()`** va être exécutée sur le device
  - **`add()`** va être appelée à partir du host

- Puisque **`add()`** s'exécute sur le device, **`a`**, **`b`** et **`c`** doivent pointer sur des zones de la mémoire du GPU. On doit donc pouvoir allouer de la mémoire sur le GPU.

Gestion de la Mémoire (Host/Device)

- Les mémoires du Host et Device sont des entités séparées
  - *Les pointeurs du Device* pointent sur la mémoire du GPU

    Ils peuvent être passés vers/à partir du code host

    Ils ne peuvent pas être déréférencés (désalloués) dans le code host

  - *Les pointeurs Host* pointent sur la mémoire du CPU

    Ils peuvent être passés vers/à partir du code device

    Ils ne peuvent pas être déréférencés (désalloués) dans le code device

- La mémoire du dévice peut être manipulée par de simples fonctions de l'API CUDA
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Elles sont similaires aux fonctions C équivalentes

    `malloc()`, `free()`, `memcpy()`

Code CUDA complet de l'addition de deux entiers

```
__global__ void add(int *a, int *b, int *c)  {
    *c = *a + *b;
}
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2; b = 7;
        // Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
    }
```

Calcul Parallèle avec CUDA C

- Nous avons jusque là vu comment invoquer la fonction noyau pour exécuter un certain traitement sur l'accélérateur en utilisant la syntaxe <<<1,1>>>.
- L'appel de la fonction noyau avec la syntaxe <<<1,1>>> ne permet de lancer qu'une seule instance du calcul, aucun parallélisme n'est possible !
- On a intérêt de lancer plusieurs instances de la fonction noyau en même temps pour pouvoir exécuter plusieurs traitements en parallèle.
- Généralement, il s'agit du parallélisme de données. C'est le même traitement qui est appliqué à des données différentes.
- Il est possible de lancer plusieurs instances de la fonction noyau en changeant la manière dont elle est appelée.

Calcul Parallèle avec CUDA C en utilisant les blocs

- Un appel avec la syntaxe <<<N,1>>> permet de lancer N instances de la fonction noyau en parallèle.
- Terminologie :
  - Chaque invocation de la fonction noyau est appelée Bloc. C-à-d, un bloc est associé à chaque instance de la fonction noyau.
  - L'ensemble des blocs est appelé grille.
- Chaque instance de la fonction noyau peut référencer l'indice de son bloc en utilisant la syntaxe **`blockIdx.x`** (**`.x`** signifie qu'on utilise la représentation des blocs sur une dimension)
- Les différentes valeurs de **`blockIdx.x`** peuvent être utilisées pour accéder aux différents éléments d'un tableau,
- Nous allons utiliser les indices de blocs (**`blockIdx.x`**) pour réaliser l'addition (en parallèle) de deux vecteurs d'entiers.

Addition de deux vecteurs avec CUDA C

- Chaque bloc (ou instance de la fonction noyau) va s'occuper de l'addition des deux valeurs des vecteurs ayant pour indices le numéro du bloc en cours.
- Les différentes additions se font en parallèle sur les unités de calculs de l'accélérateur.
- Nou allons lancer autant de blocs (instances de la fonction noyau) qu'il y a d'éléments dans un vecteur

Bloc 0

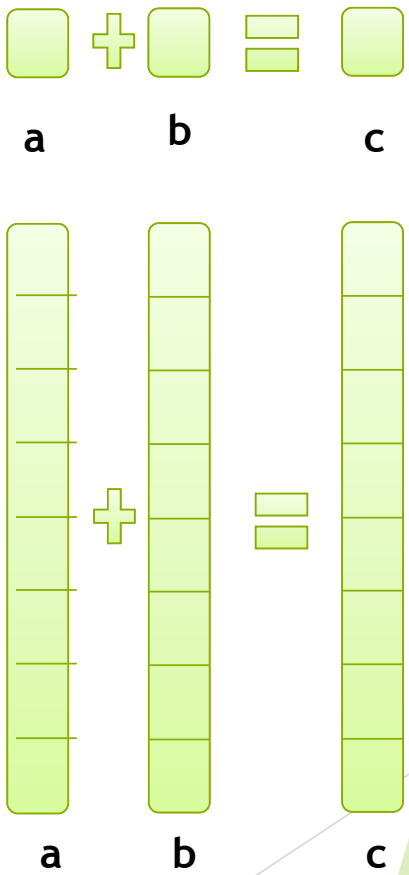
```
c[0]  = a[0] + b[0];
```

Bloc 1

```
c[1]  = a[1] + b[1];
```

Bloc 2

```
c[2]  = a[2] + b[2];
```



Programme CUDA C pour l'addition de deux vecteurs

```
#define N 512
    __global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
    }
int main(void) {
        int *a  *b  *c        // host copies of a, b, c
        int *d_a, *d_b, *d_c;// device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c and setup input
values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);
```

Programme CUDA C pour l'addition de deux vecteurs (2)

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Calcul Parallèle avec CUDA C en utilisant les Threads

- Un appel avec la syntaxe <<<N,1>>> permet de lancer N instances de la fonction noyau en parallèle.
- Chaque instance de la fonction noyau est désignée par un bloc.
- Chaque bloc contient un seul thread. La syntaxe <<<N,1>>> désigne donc N blocs de 1 thread chacun.
- Au lieu de cela, on peut lancer 1 bloc de N threads, Ceci est possible en invoquant la fonction noyau avec la syntaxe <<<1,N>>>
- Chaque instance de la fonction noyau est désignée par un thread.
- Chaque instance de la fonction noyau peut référencer l'indice de son thread en utilisant la syntaxe **`threadIdx.x`** (**`.x`** signifie qu'on utilise la représentation de threads sur une seule dimension)

Addition de deux vecteurs en utilisant les threads

- Les différentes valeurs de **`threadIdx.x`** peuvent être utilisées pour accéder aux différents éléments d'un tableau,

- Nous allons utiliser les indices de blocs (**`threadIdx.x`**) pour réaliser l'addition (en parallèle) de deux vecteurs d'entiers.

- Question : quelle est la manière la plus avantageuse entre les deux ?

- Réponse : en l'absence de communications entre les threads, les deux manières sont équivalentes. Mais en présence de communications, on doit utiliser la deuxième manière.

- Pourquoi ? pour que les différents threads puissent communiquer et coopérer afin de réaliser une certaine tâche.

Addition de deux vecteurs en utilisant les threads

- Nous changeons le programme précédent pour utiliser les threads parallèles au lieu des blocs parallèles,

- Pour ce faire, il suffit de changer la fonction noyau comme suit :

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- Et de changer la manière dont elle est appelée comme suit :

```
// Launch add() kernel on GPU with N threads
        add<<<1,N>>>(d_a, d_b, d_c);
```

Calcul Parallèle en utilisant les blocs et les Threads

- Les deux manière d'exprimer le parallélisme vues précédemment ont un inconvénient commun qui réside dans le fait qu'elle sont limitées en terme du nombre maximale de threads qu'on peut implémenter

- Pour des contraintes de conception matérielles, les nombres maximum de blocs et de threads par bloc sont limités.

- Le nombre Max de Blocs est 65535 et le nombre Max de threads par bloc est 512 ou 1024 pour les GPU de capacité supérieure ou égale à 2.0

- Autrement dit, pour notre exemple d'addition de deux vecteurs, il ne faut pas que la taille du vecteur (N) dépasse 65535 dans le premier cas et 1024 dans le deuxième

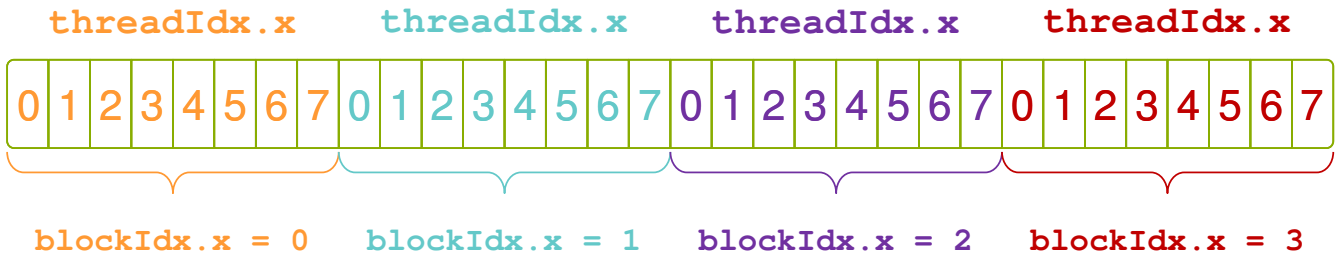
- Pour remédier à cela, on peut combiner l'utilisation des blocs et threads parallèles

Indexation des tableaux en utilisant les threads et les blocs

- Le calcul de l'indice d'un élément d'un tableau est calculé en fonction des variables prédéfinies **`blocIdx.x`** et **`threadIdx.x`** comme suit :

- `int index = threadIdx.x + blockIdx.x * M;` (`M` est le nombre de threads par bloc)

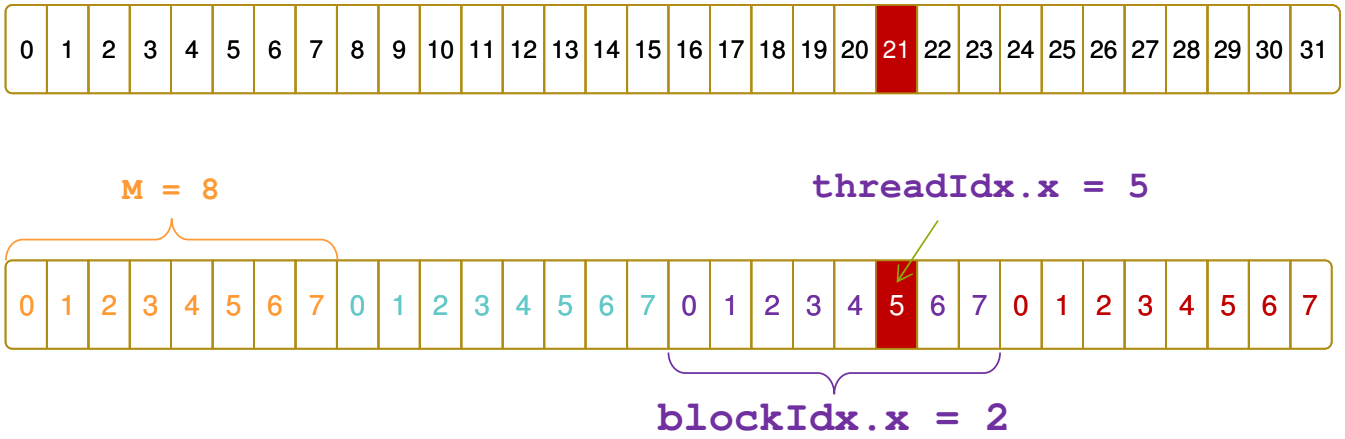
- Exemple : (M=8)



Indexation des tableaux en utilisant les threads et les blocs

- Par exemple : quel thread va opérer sur l'élément en rouge (`M=8`) ?

- C'est le thread numéro 5 du bloc numéro 2




```
int index = threadIdx.x + blockIdx.x * M;
          =       5       +      2       * 8;
          = 21;
```

Indexation des tableaux en utilisant les threads et les blocs

- On peut utiliser la variable prédéfinie **`blockDim.x`** pour référencer au nombre (maximum) de threads contenus dans un bloc.

- L'indice d'un élément du vecteur est ainsi donné par :

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- La version combinée de la fonction noyau qui utilise les blocs et threads parallèles pour réaliser l'addition de deux vecteurs est donnée par :

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- Il faudra aussi modifier le fonction **`main()`** comme suit :

Indexation des tableaux en utilisant les threads et les blocs

```
#define N (2048*2048)
   #define THREADS_PER_BLOCK 512
   int main(void) {
       int *a, *b, *c;                         // host copies of a, b, c
       int *d_a, *d_b, *d_c;          // device copies of a, b, c
       int size = N * sizeof(int);

       // Alloc space for device copies of a, b, c
       cudaMalloc((void **)&d_a, size);
       cudaMalloc((void **)&d_b, size);
       cudaMalloc((void **)&d_c, size);

       // Alloc space for host copies of a, b, c and setup input values
       a = (int *)malloc(size); random_ints(a, N);
       b = (int *)malloc(size); random_ints(b, N);
       c = (int *)malloc(size);
```

Indexation des tableaux en utilisant les threads et les blocs

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Manipulation de vecteurs de taille quelconque

- Dans un grand nombre d'applications, la taille des vecteurs n'est pas un multiple du nombre de threads par bloc `blockDim.x`

- Afin d'éviter l'accès en dehors du vecteur (avec un indice supérieur ou égal à sa taille), nous devons effectuer un changement dans la fonction noyau ainsi que dans la manière dont elle est appelée comme suit :

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

add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c);
```

- Exemple 1 : pour `M=128` et `N=127, N/M = 0` (Lancer 0 bloc !), mais `(N+M-1)=1`

Manipulation de vecteurs de taille quelconque

- Exemple 2 : pour **M=128** et **N=40,<<<(N + M-1) / M,M>>> = <<<1,128>>**

  => on lance 128 threads, alors qu'on a seulement 40 éléments. Alors, on ne doit rien faire lorsque (**indexe >= N**)

- Dans le cas ou la taille du vecteur dépasse le nombre maximal de threads autorisés, c-à-d, 65535 x 1024 (ce qui est déjà énorme), nous devons effectuer un autre changement dans la fonction noyau, comme suit :

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    while (index < N)
        c[index] = a[index] + b[index];
        index += blockDim.x * gridDim.x;
}
```

C-à-d, on incrémente à chaque fois l'indice du vecteur du nombre total de threads autorisés par le GPU (nombre de threads par bloc x nombre de blocs),

Coopération entre les threads

- Nous avons vu que qu'on peut se contenter des blocs pour faire des calculs parallèles et que l'utilisation des threads introduit un degré de complexité supplémentaire.
- Alors quel est l'intérêt d'utiliser les threads ?
- Contrairement aux blocs, les threads disposent de mécanismes qui leur permettent de :
  - Communiquer entre eux pour coopérer à la réalisation d'une certaine tâche,
  - Synchroniser leurs actions pour éviter des comportements imprévisibles à cause de l'accès concurrent aux données «race condition»
- Dans l'exemple vu précédemment (addition de deux vecteurs), on n'avait pas besoin de faire communiquer les threads entre eux (les traitements étaient totalement indépendants les uns des autres).
- Dans ce qui suit, nous allons étudier un exemple dans lequel les threads ont besoin de communiquer entre eux pour calculer une seule valeur qui est le produit scalaire de deux vecteurs.

Partage de données entre threads

- Nous avons précédemment utilisé les threads pour contourner les limites matérielles du nombre de blocs qu'il est possible de créer.
- CUDA C met à disposition une zone de mémoire appelée mémoire partagée.
- On peut qualifier une déclaration de variable par le mot-clé `__shared__` afin qu'elle soit stockée en mémoire partagée.
- CUDA C crée une copie des variables partagée pour chaque bloc qu'on lance sur le GPU.
- Chaque thread d'un bloc partage les instances des variables partagées avec les threads du même bloc. Les autres threads des autres blocs ne peuvent ni voir ni modifier ces instances.
- Ceci fourni un excellent moyen de communication et de collaboration entre les threads du même bloc.

Partage de données entre threads et synchronisation

- Les tampons de mémoire partagée étant implémentés physiquement sur le GPU. Ce qui permet de les utiliser comme des caches gérés par le logiciel propre à chaque bloc.
- L'utilisation de la mémoire partagée comme cache permet d'accélérer le temps d'accès aux données partagées par rapport au temps d'accès aux données situées sur la DRAM externe au circuit.
- Mais pour que les threads puissent communiquer, nous avons besoin d'un mécanisme de synchronisation entre eux afin d'éviter le problème d'accès concurrent au données.
- Si par exemple un thread B a besoin de lire une donnée x qu'un autre threads A va écrire, alors B ne doit pas commencer la lecture tant que A n'a pas encore fini l'écriture.
- Sous CUDA C, on peut synchroniser les threads en appelant la fonction `__syncthreads()` qui garantit que chaque thread du bloc a terminé les instructions qui précèdent cet appel.

Notion de réduction

- Dans beaucoup d'applications, un seul résultat est obtenu en traitant un grand ensemble de données.

- Par exemple, dans le produit scalaire de vecteurs, on fait autant de multiplications qu'il y a de données dans un vecteur et on fait la somme de toutes les multiplications.

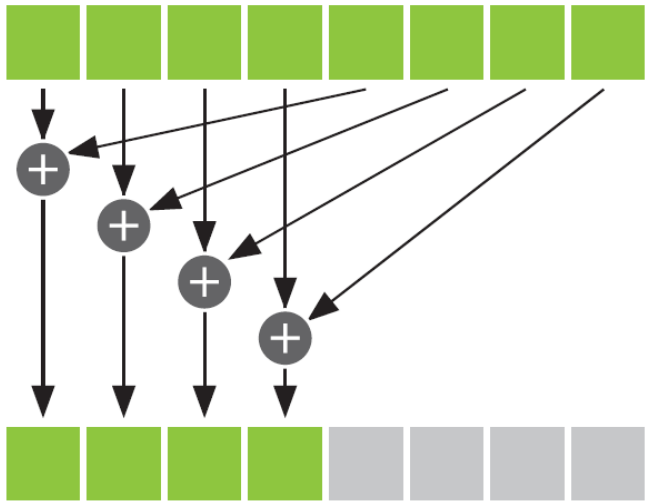
- Lorsqu'un algorithme parallèle est utilisé, chaque thread s'occupe du traitement d'un sous ensemble de données et calcule un résultat partiel.
- Les différents résultats partiels sont ensuite combinés pour en produire un seul résultat global.

- Ceci peut être réalisé d'une manière séquentielle en parcourant une boucle, ou d'une manière parallèle en utilisant une opération de réduction où un certain nombre de threads sont utilisés pour accélérer la production du résultat global

Exemple de réduction (calcul de la somme)

- Exemple de code pour réaliser une réduction : le nbr d'éléments doit être = 2^n

- Chaque thread calcule la somme de deux éléments du vecteur et stocke le résultat dans la première moitié du tableau, et ainsi de suite jusqu'à n'obtenir qu'une seule valeur stockée dans le premier élément du tableaux,



```
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
```

Produit Scalaire de deux vecteurs (Kernel)

```
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
            imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
```

Produit Scalaire de deux vecteurs (Kernel)

```
    // set the cache values
    cache[cacheIndex] = temp;
    // synchronize threads in this block
    __syncthreads();
    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
       i /= 2;
    }
    if (cacheIndex == 0)
       c[blockIdx.x] = cache[0];
}
```

Produit Scalaire de deux vecteurs (changement main)

```
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                          blocksPerGrid*sizeof(float) ) );

dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                        dev_partial_c );

c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
```

Mesure de performances de programmes CUDA C

- Il est très important de pouvoir mesurer les performances de programmes CUDA C pour pouvoir comparer différentes versions de codes, et savoir si une version donnée a amélioré le temps d'exécution ou au contraire elle l'a dégradé.

- La mesure de performances d'une partie de du code est possible grâce au concept d'événement CUDA.

- Un événement CUDA est essentiellement une étiquette temporelle du GPU enregistrée à un instant précis, choisi par l'utilisateur.

- La mesure des performances d'une partie de code CUDA se fait en utilisant les fonctionnalités de CUDA, comme suit :

✓ Déclaration des événements :

```
cudaEvent_t start, stop;
```

✓ Création des événements :

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

Mesure de performances de programmes CUDA C

- Enregistrement des événements : Les deux événements sont enregistrés autour de la région dont on souhaite mesurer les performances

```
cudaEventRecord(start, 0);
// code à mesurer
cudaEventRecord(stop, 0);
```

- Synchronisation CPU/GPU

```
cudaEventSynchronize(stop);
```

- Calcul du temps écoulé entre les deux instants

```
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

- Destruction (libération) des événements :

```
cudaEventDestroy(&start);  cudaEventDestroy(&stop);
```

Atomicité en CUDA C

- Certaines situations extrêmement simples à traiter avec des applications mono-threads posent un sérieux problème lorsque l'on essaie de les implémenter sur une machine massivement parallèle.

- Par exemple, considérant la situation ou deux threads A et B ont besoin d'incrémenter la valeur stockée dans x. tous les deux doivent exécuter les trois opérations suivantes :
  1. Lire la valeur de x;
  2. Ajouter 1 à la valeur obtenue à l'étape 1;
  3. Ecrire le résultat dans x.

- Supposons que la valeur initiale de x est 7, alors à la fin d'exécution des threads A et B, on peut obtenir 9 ou bien 8, selon l'ordre dont les trois opérations sont effectuées par les threads A et B.

- On dit qu'il y a un comportement indéterministe du programme.

Atomicité en CUDA C

- Pour éviter ce genre de problèmes, il faut exécuter les trois opérations précédentes en un seul coup (sans interruption).
- Une fois un thread a commencé la procédure d'incrémentation de la variable x, aucun autre thread n'aura le droit d'accéder à x, jusqu'à ce que le premier thread aura terminé le processus d'incrémentation.
- On dit que l'opération d'incrémentation doit se faire d'une manière atomique.
- CUDA propose un certain nombre d'opérations atomiques en utilisant une syntaxe très simple, mais l'utilisation doit être faite d'une manière rigoureuse.
- Par exemple : l'incrémentation de x peut être réalisée en utilisant l'opération atomique : `atomicAdd (&x,1)`
- Nous allons expliquer le concept de l'atomicité en utilisant un algorithme très utile en traitement d'image : Calcul de l'histogramme.

Calcul de l'histogramme (1)

```
int main( void ) {
    unsigned char *buffer = (unsigned char*) big_random_block( SIZE );
    cudaEvent_t start, stop;
    cudaEventCreate( &start );
    cudaEventCreate( &stop );
    cudaEventRecord( start, 0 );
   // allocate memory on the GPU for the file's data
    unsigned char *dev_buffer;
    unsigned int *dev_histo;
    cudaMalloc( (void**)&dev_buffer, SIZE );
    cudaMemcpy( dev_buffer, buffer, SIZE, cudaMemcpyHostToDevice );
    cudaMalloc( (void**)&dev_histo, 256 * sizeof( long ));
    cudaMemset( dev_histo, 0, 256 * sizeof( int ));
```

Calcul de l'histogramme (2)

```
cudaDeviceProp prop;
cudaGetDeviceProperties( &prop, 0 );
int blocks = prop.multiProcessorCount;
histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );
unsigned int histo[256];
cudaMemcpy( histo, dev_histo,256*sizeof( int ),cudaMemcpyDeviceToHost);
// get stop time, and display the timing results
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );
printf( "Time to generate: %3.1f ms\n", elapsedTime );
```

Calcul de l'histogramme (3)

```
    long histoCount = 0 ;
    for (int i=0; i<256; i++)
         histoCount += histo[i];
    printf( "Histogram Sum: %ld\n", histoCount );
    // verify that we have the same counts via CPU
    for (int i=0; i<SIZE; i++)
         histo[buffer[i]]--;
    for (int i=0; i<256; i++) {
        if (histo[i] != 0)
           printf( "Failure at %d!\n", i );
    }
    cudaEventDestroy( start ); cudaEventDestroy( stop );
    cudaFree( dev_histo ); cudaFree( dev_buffer ); free( buffer );
    return 0;
}
```

Calcul de l'histogramme : La fonction Noyau sur Mémoire Globale

```
#define SIZE (100*1024*1024)
__global__ void histo_kernel( unsigned char *buffer,
                              long size, unsigned int *histo ) {

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1 );
        i += stride;
    }
}
```

- ✓ L'utilisation de la mémoire globale influe sur les performances du programme à cause des filles d'attente d'un grand nombre de threads sur les instructions atomiques.

Calcul de l'histogramme : La fonction Noyau sur Mémoire Globale et Mémoire Partagée

```
__global__ void histo_kernel( unsigned char *buffer,
                              long size, unsigned int *histo ) {
   __shared__ unsigned int temp[256];
   temp[threadIdx.x] = 0;
   __syncthreads();
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int offset = blockDim.x * gridDim.x;
   while (i < size) {
      atomicAdd( &temp[buffer[i]], 1);
      i += offset;
   }
   __syncthreads();
   atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}
```

Traitement des erreurs en CUDA C

- Il est possible que des erreurs se produisent au moment de l'appel des différentes fonctions prédéfinies CUDA. Il convient de les traiter d'une manière convenable en définissant une fonction Macro comme suit :

```
static void HandleError( cudaError_t err, const char *file, int line
) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
file, line );
        exit( EXIT_FAILURE );
    }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))
```

- Chaque appel d'une function CUDA est ensuite effectué en appellant la function Macro HANDLE_ERROR comme dans l'exemple suvant :

```
HANDLE_ERROR(cudaMalloc((void **)&d_a, size));
```

Advanced Atomics

- We will show the advanced Atomics using vector dot product we have seen before.
- Recall that the algorithm computed the dot product of two input vectors by doing the following:

1- Each thread in each block multiplies two corresponding elements of the input vectors and stores the products in shared memory.

2. Although a block has more than one product, a thread adds two of the products and stores the result back to shared memory. Each step results in half as many values as it started with (this is where the term reduction comes from)

3. When every block has a fnal sum, each one writes its value to global memory and exits.

4. If the kernel ran with N parallel blocks, the CPU sums these remaining N values to generate the fnal dot product.

Advanced Atomics

- In step 4 of the algorithm, although it doesn't involve copying much data to the host or performing many calculations on the CPU, moving the computation back to the CPU to finish is indeed as awkward as it sounds.

- It's more than an issue of an awkward step to the algorithm or the inelegance of the solution. Consider a scenario where a dot product computation is just one step in a long sequence of operations.

- If you want to perform every operation on the GPU because your CPU is busy with other tasks or computations, you're out of luck. As it stands, you'll be forced to stop computing on the GPU, copy inter- mediate results back to the host, fnish the computation with the CPU, and fnally upload that result back to the GPU and resume computing with your next kerne

Advanced Atomics

- Since we have already used an atomicAdd() operation in the histogram operation, this seems like an obvious choice.
- Unfortunately, prior to compute capability 2.0, atomicAdd() operated only on integers. Although this might be fine if you plan to compute dot products of vectors with integer components,
- It is significantly more common to use foating-point components. However, the majority of NVIDIA hardware does not support atomic arithmetic on foating-point numbers!
- The Atomic computation may also be more than just one operation, it could be a some sequence of instructions,
- So we do need another way to perform Atomic computations

Atomic Locks

- The basic idea is that we allocate a small piece memory to be used as a mutex.
- The mutex will act like something of a traffic signal that governs access to some resource. The resource could be a data structure, a buffer, or simply a memory location we want to modify atomically.
- When a thread reads a 0 from the mutex, it interprets this value as a “green light” indicating that no other thread is using the memory. Therefore, the thread is free to lock the memory and make whatever changes it desires, free of interference from other threads.
- To lock the memory location in question, the thread writes a 1 to the mutex. This 1 will act as a “red light” for potentially competing threads. The competing threads must then wait until the owner has written a 0 to the mutex before they can attempt to modify the locked memory.

Atomic Locks

- A simple code sequence to accomplish this locking process might look like this:

```
void lock( void ) {
        if( *mutex == 0 ) {
             *mutex = 1; //store a 1 to lock
        }
}
```

- Unfortunately, there's a problem with this code. What happens if another thread writes a 1 to the mutex after our thread has read the value to be zero? That is, both threads check the value at mutex and see that it's zero. They then both write a 1 to this location to signify to other threads that the structure is locked and unavailable for modification.
- After doing so, both threads think they own the associated memory or data structure and begin making unsafe modifications. Catastrophe ensues!

Atomic Locks

- The operation we want to complete is fairly simple: We need to compare the value at mutex to 0 and store a 1 at that location if and only if the mutex was 0.
- To accomplish this correctly, this entire operation needs to be performed atomically so we know that no other thread can interfere while our thread examines and updates the value at mutex.
- In CUDA C, this operation can be performed with the function atomicCAS(), an atomic compare-and-swap. The function atomicCAS() takes a pointer to memory, a value with which to compare the value at that location, and a value to store in that location if the comparison is successful.
- Using this operation, we can implement a GPU lock function as follows: