Programmation GPU

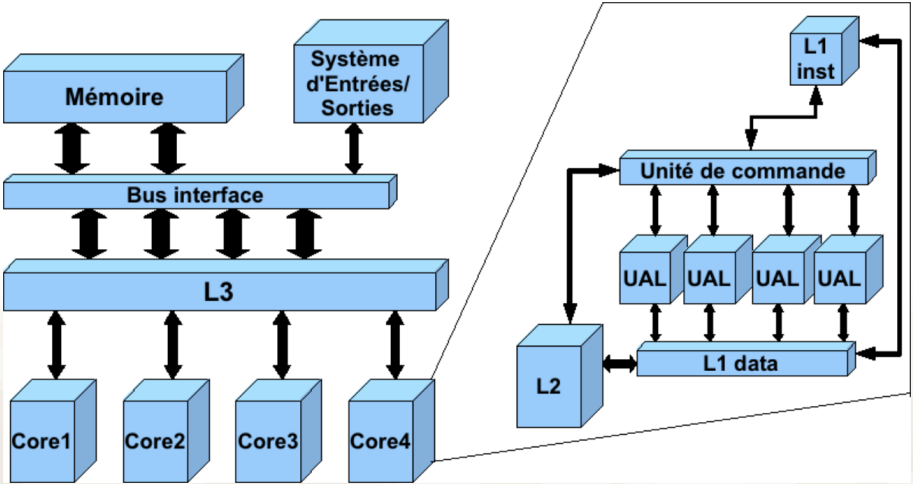
David Grimbichler
david.grimbichler@uca.fr

Mésocentre Université Clermont Auvergne

Sommaire

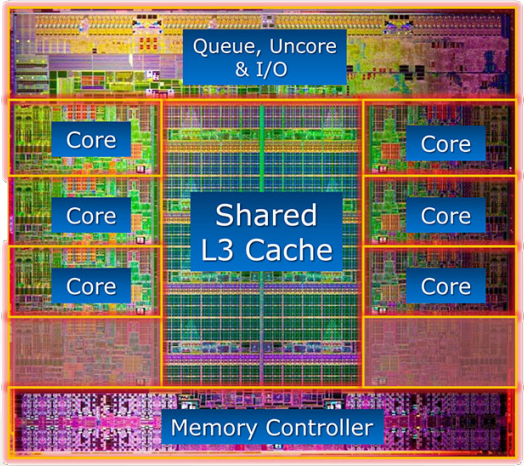
1. Rappels
   - Rappels : différences CPU vs GPU
   - Rappels : comparaison en chiffres, ordres de grandeur
   - Rappels : parallélisme
2. Architecture GPU
   - Exemple : Fermi
   - Les mémoires
3. Programmation GPU
   - Introduction CUDA
   - Organisation des threads en CUDA
   - La mémoire en CUDA
   - Les threads sur le GPU
   - Debug CUDA
   - Profiling CUDA
   - Programmer en CUDA
   - Exemple et analyse mémoire partagée : produit scalaire
   - Deep learning et CUDA

Rappel : CPU et architecture classique

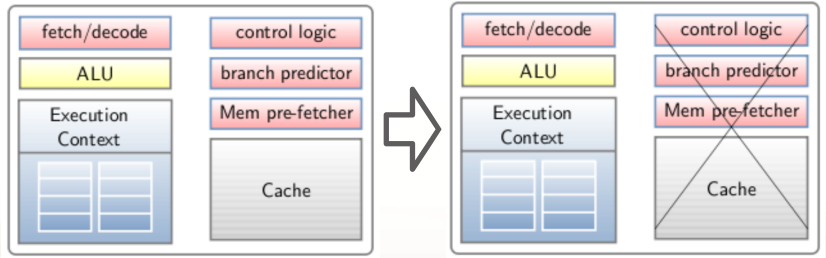


Rappel : Exemple die CPU

Intel® Core™ i7-3960X Processor Die Detail



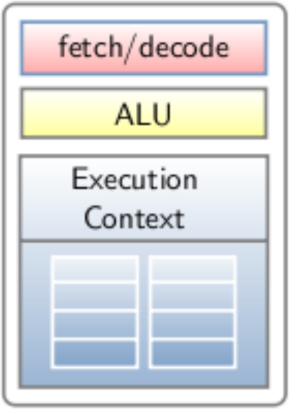
Comparaison par rapport à un CPU



On enlève les composants qui permettent de gérer le flux d'instructions de façon optimale.

La base du GPU

On obtient un coeur d'un GPU :

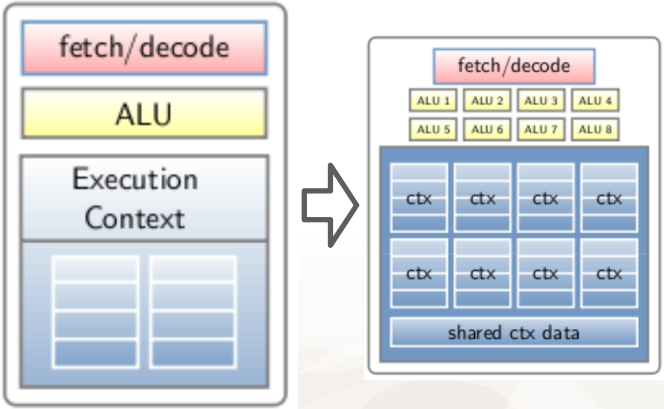


Base du GPU

On met plein de ces coeurs sur un die :

La base du GPU

Partage du flux d'instructions :



Le GPU

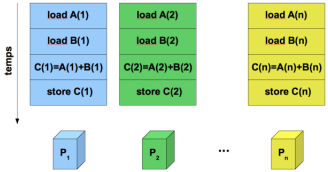
- GPU : ensemble de N petites machines SIMD indépendantes et partageant une mémoire globale (N multiprocesseurs, N=16 ici)
- Un multiprocesseur est 1 petite machine SIMD avec :
  - k ALU (Stream processor) synchronisés (ici k=8)
  - 1 décodeur d'instructions
  - 3 mémoires partagées entre toutes les ALU (dont 2 cache)
  - 8192 registres distribués entre les ALU
- 128 coeurs

CPU vs GPU

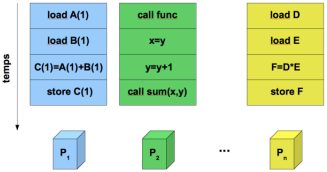
CPU : optimisé pour exécuter rapidement une série de tâches de tout type. Optimisé pour une faible latence.
GPU : optimisé pour exécuter des opérations simples sur de très gros volumes de données. Optimisé pour une forte bande passante.

GPU : majoritairement SIMD

SIMD architecture // , toutes les unités de traitement exécutent la même instruction à un cycle d'horloge donnée, chaque unité peut opérer sur des données différentes, exécution déterministe. (GPU)



MIMD architecture la plus courante. Chaque unité de traitement peut gérer un flot d'instructions différent. Chaque unité peut opérer sur un flot de données différent. L'exécution peut être synchrone ou asynchrone, déterministe ou non-déterministe. (CPU)



Sommaire

1 Rappels
- Rappels : différences CPU vs GPU
- Rappels : comparaison en chiffres, ordres de grandeur
- Rappels : parallélisme

2 Architecture GPU

3 Programmation GPU

Quelques chiffres

Exemple de matériel grand public en 2007 :

	GeForce 8800 GT	Intel Core 2 Duo
Nombre de coeurs	112	2
Fréquence	1,5GHz	3GHz
Performance théorique	367 GFlops	32 GFlops
Bande passante mémoire	86,4GB/s	8,4GB/s

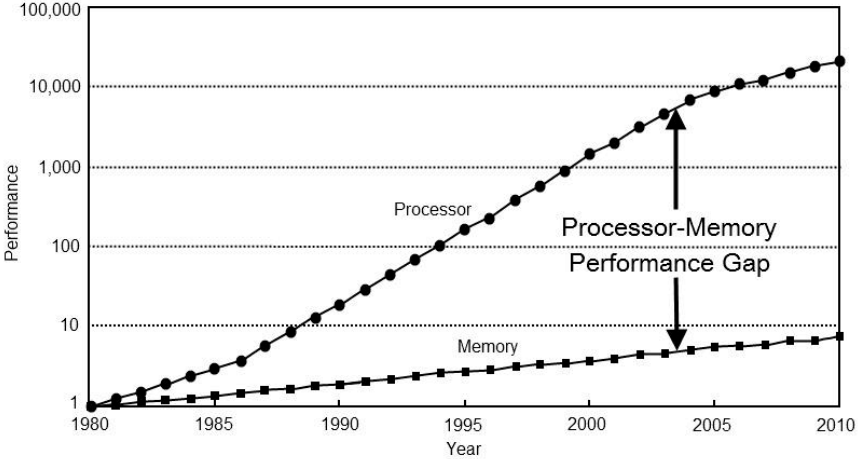
Quelques chiffres

Exemples de matériel :

	GeForce GTX 1080 (2007)	Intel Xeon Platinum 8180 (2007)	GeForce RTX 3090 (2020)	AMD Ryzen Threadrip-per 3990X (2020)
Nombre de coeurs	3584	28	10496	64
Fréquence	1,5GHz	3,6GHz	1,4GHz	4GHz
Performance théorique	10600 GFlops	1500 GFlops	35500 GFlops	3700 GFlops
Bande passante mémoire	330GT/s	10,4GT/s	556GT/s	24GT/s
Prix	600 euros	10000 euros	2500 euros	4700 euros

- présent dans beaucoup de PC : marché de masse
- adapté au massivement parallèle (milliers de threads)
- programmable via des API spécifiques

Problème : Processor-Memory performance gap



Conclusion sur les architectures

- Vers une augmentation du **nombre de coeurs** plutôt que de la fréquence d'horloge (problème finesse de gravure, échauffement)
- Programmation de plus en plus complexe
  - nécessité d'une **bonne connaissance des architectures**, notamment au niveau de la mémoire
  - pas de croissance des performances au niveau du coeur $\Rightarrow$ **parallélisation** obligatoire pour gagner des performances
  - Nécessité d'élaborer des **algorithmes** et des programmes capables d'exploiter un grand nombre de processeurs
  - Nécessité d'exploiter le **parallélisme** aux différents niveaux du hardware
  - Programmation sur des architectures hybrides avec différents types de processeurs
- Exploitation des GPU : code extrêmement parallélisé, capable d'utiliser TOUS les threads

Sommaire

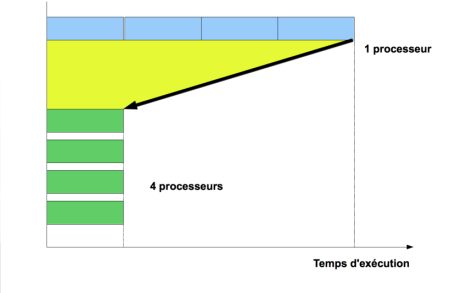
1 Rappels
- Rappels : différences CPU vs GPU
- Rappels : comparaison en chiffres, ordres de grandeur
- Rappels : parallélisme

2 Architecture GPU

3 Programmation GPU

Séquentiel vs parallèle

- **Exécutions séquentielles des programmes** : le processeur n'exécute les instructions que l'une après l'autre, en passant éventuellement d'un processus à un autre via des interruptions en simulant le multitâche.
- **Parallélisation** : Ensemble de techniques logicielles et matérielles permettant l'exécution simultanée de séquences d'instructions indépendantes, sur des processeurs différents.



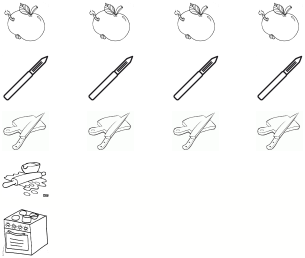
Séquentiel vs parallèle : calculons... (1)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



---

43 minutes à 1 personne

Combien à 2 personnes ?

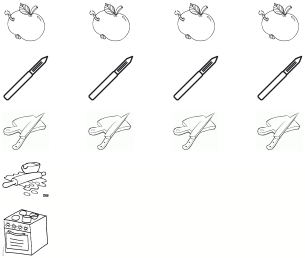
Séquentiel vs parallèle : calculons... (2)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



---

37 minutes à 2 personnes

Combien à 4 personnes ?

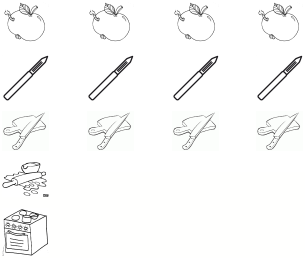
Séquentiel vs parallèle : calculons... (3)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



---

35 minutes à 4 personnes

Combien à 3 personnes avec un seul couteau et un seul économe ?

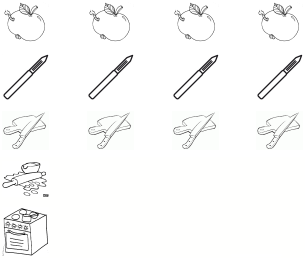
Séquentiel vs parallèle : calculons... (4)

4 x 1 minutes

4 x 1 minutes

1 x 5 minutes

1 x 30 minutes



---

35 min à 3 + 1 couteau + 1 économe

Pourquoi paralléliser ?

- Traitement de problèmes scientifiques constituant de grands challenges, tels que : météo et climat, biologie (génome), géophysique (activité sismique), réactions chimique et réactions nucléaires, ...
- Mais aujourd'hui également pour des applications commerciales : Bases de données parallèles, exploration pétrolière, moteurs de recherche web, réalité virtuelle, ...
- Rendre une meilleure expérience utilisateur, plus fluide, plus rapide (traitement d'images sur les smartphones)

Paralléliser pour :

- Traiter des problèmes plus grands et/ou plus complexes.
- Exploiter les architectures actuelles.
- L'énergie est notre avenir, économisons-la.

Sommaire

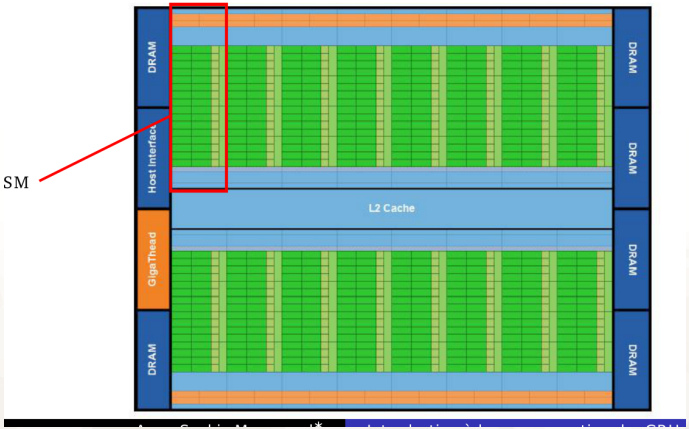
1 Rappels

2 Architecture GPU
- Exemple : Fermi
- Les mémoires

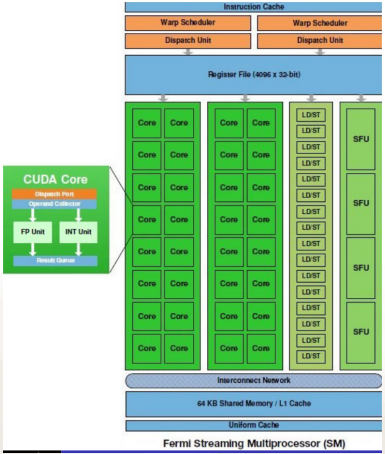
3 Programmation GPU

Exemple : architecture Fermi

Jusqu'à 16 Streaming Multiprocessors (SM) de 32 cœurs CUDA soit 512 cœurs.



SM Streaming Multiprocessor



- Regroupe 32 cœurs ayant des unités de calcul 64 bits.
- Ordonnance les threads par paquets (via le Warp Scheduler).
- Dispose de registres, de mémoire partagée et de caches.

Sommaire

1 Rappels

2 Architecture GPU
- Exemple : Fermi
- Les mémoires

3 Programmation GPU

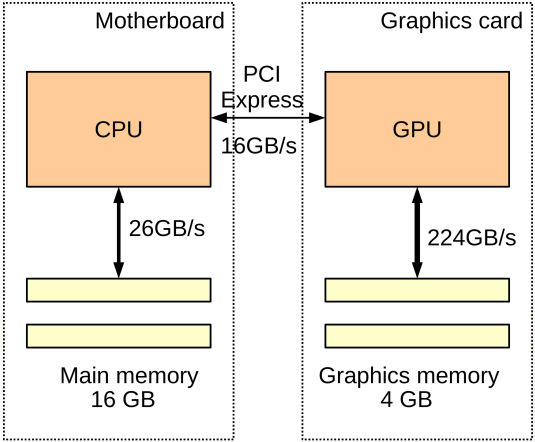
Types de mémoires

- Mémoires sur la puce
  - registres, très rapides (RW), 32 bits (32768 par SM) ;
  - mémoire partagée (RW), jusqu'à 48 KiB par SM.
- Mémoires hors de la puce :
  - mémoire globale (RW), quelques GiB, lente (quelques centaines de cycles d'horloge) ;
  - mémoire de texture (RO) ;
  - mémoire de constantes (RO) : accès rapide ;
  - mémoire locale (RW).

Transferts mémoire CPU-GPU

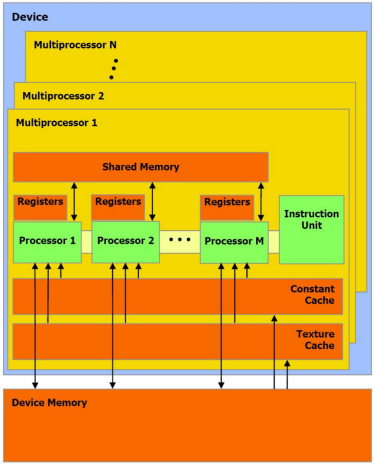
- Le GPU a sa propre mémoire (mémoire globale).
  Bande passante (BP) de l'ordre de 100 GiB/s.
  Exemple : carte Tesla 2050 avec RAM à 1500 MHz, BP = 144 GiB/s.
- En comparaison, la BP mémoire pour un CPU est de l'ordre de 10 GiB/s.
  Exemple : proc. Nehalem avec RAM à 1066 MHz, BP = 25.6 GiB/s.
- Le GPU n'accède pas à la mémoire centrale du CPU.
  Les transferts entre la mémoire du CPU et celle du GPU se font via le bus Pci-express.
  BP de l'ordre de 10 GiB/s et grande latence => pénalisant !
  Exemple : Pci-express x16, Gen2, BP = 8 GiB/s.

Transferts



Example configuration:
Intel Core i7 4790, Nvidia GeForce GTX 980

Organisation mémoire



- Pour chaque SM
  - mémoire partagée (r/w)
  - cache texture (r)
  - cache constantes (r)
  - cache instructions
  - M processeurs scalaires (ALUs)
- Pour chaque processeur (ALU) : registres (r/w)
- Mémoire partagée presque aussi rapide que les registres
- Accès à la DRAM de la carte

Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

CUDA ?

CUDA : Compute Unified Device Architecture (actuellement : CUDA 10.0)

- Développé par Nvidia depuis 2007
- Fait référence à l'architecture et au modèle de programmation
- Permet de programmer les GPU Nvidia en C/C++

Vocabulaire CUDA :

- **Host** : le CPU.
- **Device** : le GPU.
- **Kernel** : fonction s'exécutant sur le GPU (= shader).

Points clés d'un programme CUDA :

- Le programme dirige le host et le device : il contient le code pour le host et pour le device.
- Allocations de mémoire sur CPU et GPU, transferts de données dans les deux sens, ...
- Lancement du kernel sur le device par le host en précisant un contexte d'exécution.

Exemple de programme simple CUDA (C)

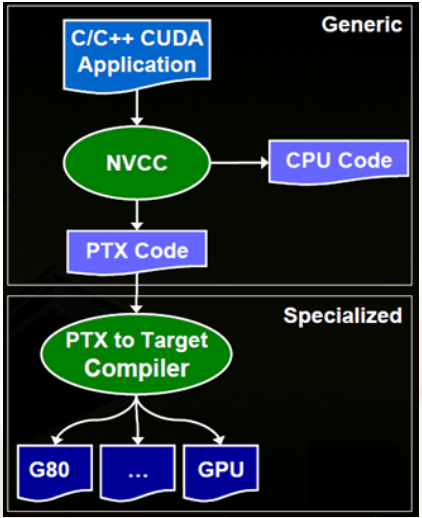
```
1  #include <stdio.h>
2  /* a simple CUDA kernel */
3  __global__ void add( int a, int b, int *c ) {
4    *c = a + b; }
5
6  int main( void ) {
7    int c;
8    int *dev_c;
9
10   // GPU device memory allocation
11   cudaMalloc( (void**)&dev_c, sizeof(int) ) ;
12
13   // perform computation on GPU
14   add<<<1,1>>>( 2, 7, dev_c );
15
16   // get back computation result into host CPU memory
17   cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost ) ;
18
19   // output result on screen
20   printf( "2 + 7 = %d\n", c );
21
22   // de-allocate GPU device memory
23   cudaFree( dev_c ) ;
24
25   return 0;
26 }
```

Exemple de programme simple CUDA (Python)

```python
import pycuda.driver as drv
import pycuda.tools
import pycuda.autoinit
import numpy
from pycuda.compiler import SourceModule

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

add = mod.get_function("add")

a = numpy.int32(2)
b = numpy.int32(7)
c = numpy.zeros_like(a)

add(a, b, drv.Out(c), block=(1,1,1))

print(c)
```

Compilateur : nvcc

```
nvcc programme.c -o programme
```



Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

Kernel

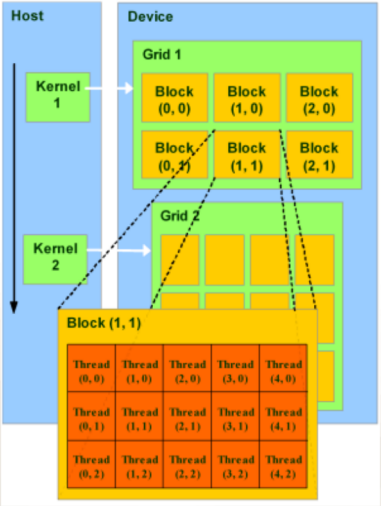
Fonction lancée depuis le host et tournant sur le device :

- de type __global__
- retournant un void
- lancée à l'aide d'une syntaxe ressemblant au C

```
myKernel <<< gridDim, blockDim >>> (parameters...);
```

gridDim : dimensions de la grille en blocs
blockDim : dimensions des blocs en threads.
Tout kernel a accès aux variables grimDim, blockDim, blockIdx, threadIdx.

Organisation des threads



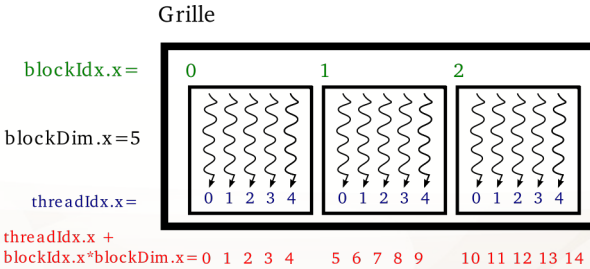
Grille 3D de blocs 3D de threads :

- Dimensions de la grille en blocs : gridDim(.x,.y,.z)
- Dimensions des blocs en threads : blockDim(.x,.y,.z)
- Indice d'un bloc dans la grille : blockIdx(.x,.y,.z)
- Indice d'un thread dans son bloc : threadIdx(.x,.y,.z)

Index d'un thread

Un thread d'une grille est identifié par un index global unique. Par exemple en 1D :

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



L'index permet d'établir une correspondance entre le thread et les données (élément *i* d'un tableau 1D par exemple).

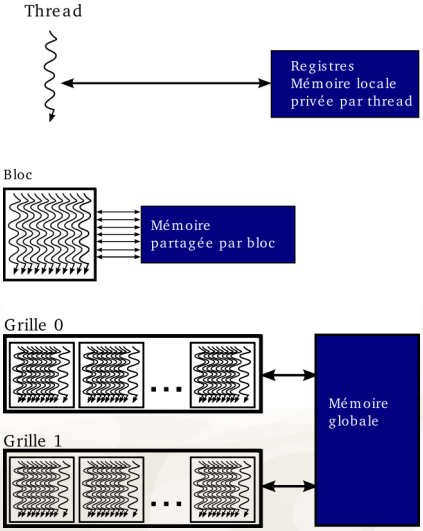
Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

Modèle mémoire



Transferts entre mémoires

Sur le GPU :

```
cudaError_t cudaMalloc(void** devPtr, size_t count);
cudaError_t cudaFree(void* devPtr);

```

À comparer au CPU :

```
void *malloc(size_t size);
void free(void *ptr);

```

Transferts entre mémoires

Copie entre mémoires du host et du device :

```
cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind);
```

copie count octets de la zone pointée par src dans celle pointée par dst.
Le sens de la copie est indiqué par kind (cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost).
Les transferts sont contrôlés par le CPU.

Exemple de copie vers le device (C)

```
1  int main( void ) {
2    // array size
3    int N = 16;
4    // host variable
5    int *a
6    // device variable
7    int *dev_a
8
9    // CPU memory allocation / initialization
10   a = (int *) malloc(N*sizeof(int));
11   for (int i=0; i<N; i++)
12   {
13     a[i]=i;
14   }
15   // GPU device memory allocation / initialization
16   cudaMalloc( (void**)&dev_a, N*sizeof(int) ) ;
17   cudaMemcpy( dev_a, a, N*sizeof(int),
18               cudaMemcpyHostToDevice ) ;
19
```

Exemple de copie vers le device (Python)

```python
import pycuda.driver as cuda
import pycuda.tools
import pycuda.autoinit
import numpy
from pycuda.compiler import SourceModule

N = 16

a = numpy.arange(0, N, 1, numpy.int32)
a_gpu = cuda.mem_alloc(a.nbytes) # (a.size * a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)

```

Exemple de kernel CUDA : vector add

CUDA :

```
/* a simple CUDA kernel */
__global__ void add( int *a, int *b, int *c,
     int n )
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if (i<n)
    c[i] = a[i] + b[i];
}
```

Host :

```
/* Version CPU pour comparaison */
void add( int *a, int *b, int *c, int n )
{
  int i ;
  for (i = 0; i < n; ++i)
  {
    c[i] = a[i] + b[i];
  }
}
```

Appel du kernel par le host :

```
int nbThreads = 8;
dim3 blockSize(nbThreads,1,1);
dim3 gridSize(N/nbThreads,1,1);
add<<<gridSize,blockSize>>>
     ( dev_a, dev_b, dev_c, N );
```

Exemple de kernel CUDA : vector add (Python)

```
mod = SourceModule("""
__global__ void add(int *a, int *b, int *c, int n)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if(i<n)
    c[i] = a[i] + b[i];
}
""")

add = mod.get_function("add")

N = 16
a = numpy.random.randint(1, 1000, N, numpy.int32)
b = numpy.random.randint(1, 1000, N, numpy.int32)
c = numpy.zeros_like(a)

nbThreads = 8;
blockSize=(nbThreads, 1, 1)
gridSize=(math.ceil(N/nbThreads), 1, 1)

add(drv.In(a), drv.In(b), drv.Out(c), numpy.int32(N),
    block=blockSize, grid=gridSize)

print(c)
print(c - (a+b))
```

Questions CUDA

- combien de threads par bloc ?
- combien de blocs ?
- dimensions maximales des blocs et de la grille ?
- quelle quantité de mémoire accessible par bloc ?
- quelle mémoire est partagée entre les threads d'un même bloc ? entre les blocs ?
- ...
- Une bonne connaissance des caractéristiques matérielles est nécessaire.

Penser aux erreurs !

Comme toujours en programmation : pensez bien à vérifier les codes de retour d'erreurs :

```
cudaError_t err = cudaMalloc((void**)&d_A, size);
if(err != cudaSuccess) {
  printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
  exit(EXIT_FAILURE);
}
```

Mieux, on définit des macros / fonctions d'aide pour tout le programme :

```
void cuda_error(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 cuda_error_check(err) (cuda_error( err, __FILE__, __LINE__ ))

...
  cuda_error_check(cudaMalloc((void**)&d_A, size));
...
```

Sommaire

1 Rappels

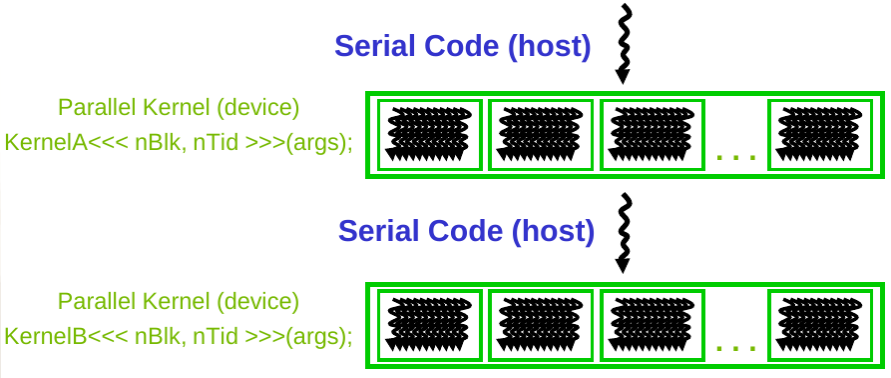
2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

Modèle d'exécution CUDA

Application s'exécutant sur une machine hétérogène : hôte (CPU) + device (GPU)

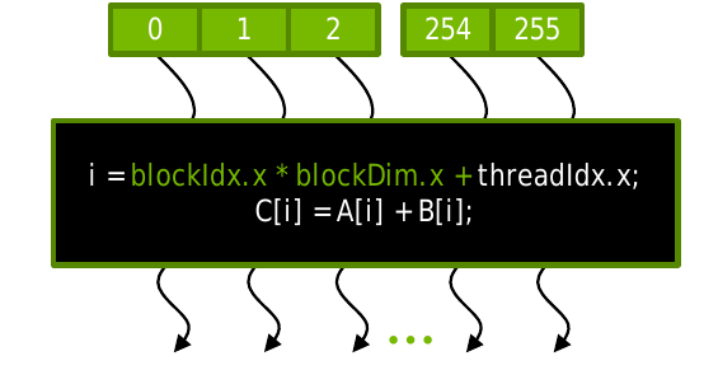
- les parties séquentielles s'exécutent sur l'hôte
- les parties parallèles s'exécutent sur le device



Tableaux de threads parallèles

Un noyau CUDA est exécuté par une grille (tableau) de threads :

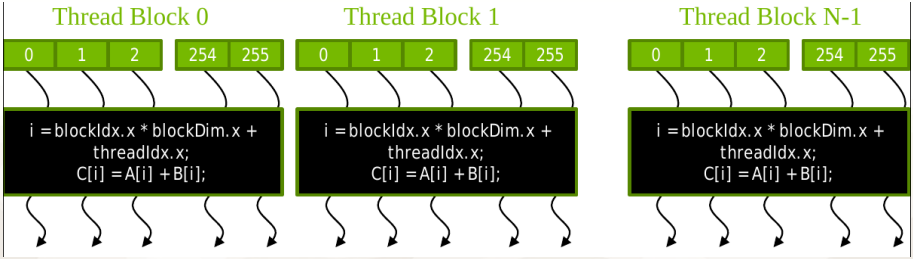
- Tous les threads d'une grille exécutent le même noyau (SPMD : Single Program Multiple Data)
- Chaque thread a des index qui sont utilisés pour calculer des adresses mémoires et prendre des décisions de contrôle



Bloc de threads : coopération

Diviser le tableau de threads en plusieurs blocs :

- les threads au sein d'un bloc coopèrent via la mémoire partagée, les opérations atomiques et les barrières de synchronisation
- les threads de différents blocs ne peuvent pas interagir



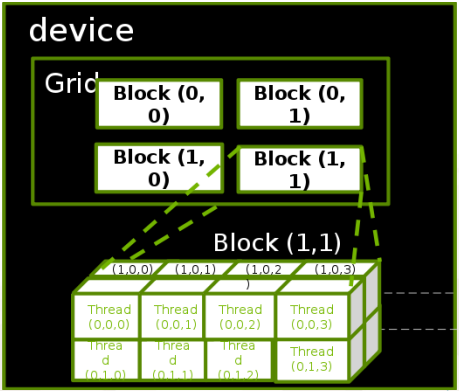
blockIdx et threadIdx

Chaque thread utilise des index pour décider sur quelle(s) donnée(s) travailler :

- blockIdx : 1D, 2D ou 3D (CUDA >= 4.0)
- threadIdx : 1D, 2D ou 3D

Pourquoi ? Pour simplifier l'adressage de la mémoire quand des données multidimensionnelles sont traitées, comme :

- traitement d'images
- résolution d'EDP sur des volumes
- mécanique des fluides en 3D
- ...



Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

Les outils de debug pour CUDA

http://on-demand.gputechconf.com/gtc/2014/presentations/S4578-cuda-debugging-command-line-tools.pdf

printf en CUDA

CUDA propose un `printf`

- même format que le `printf` C
- sortie bufferisée : flush seulement aux points de synchronisation explicites
- sortie non ordonnée : comme `printf` dans un programme multithread
- possibilité de changer la taille du buffer : `cudaDeviceSetLimit(cudaLimitPrintFifoSize, size_t size)`

Paramètres de compilation nvcc / gcc

Rappel : deux compilateurs sont utilisés :

- `nvcc` : compile le code destiné au *device*
- compilateur de l'hôte : gcc ou clang ou autre, pour le code C/C++

`NVCC` interprète certains flags de compilation de gcc,

- mais si le flag n'est pas pris en charge par nvcc, on peut passer le flag à gcc via `-Xcompiler` *`flag`*
- Exemple : `-Xcompiler -fopenmp`

Paramètres pour le debug :

- `-g` : inclure les symboles de debug pour le code gcc (*host*)
- `-G` : inclure les symboles de debug pour le code *device* (et désactive les optimisations)
- `-lineinfo` : inclure le numéro de ligne du code dans les symboles de debug

Compiler un fichier `.cpp` contenant du CUDA

```
nvcc --x=cu input.cpp ...
```

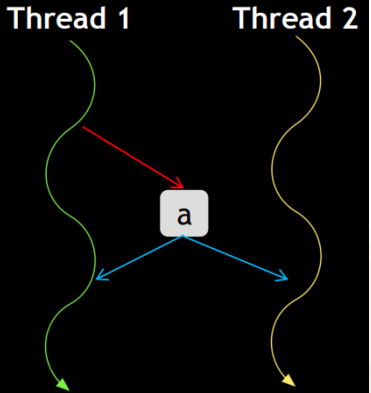
valgrind memcheck en CUDA ?

CUDA propose l'outil `cuda-memcheck` :

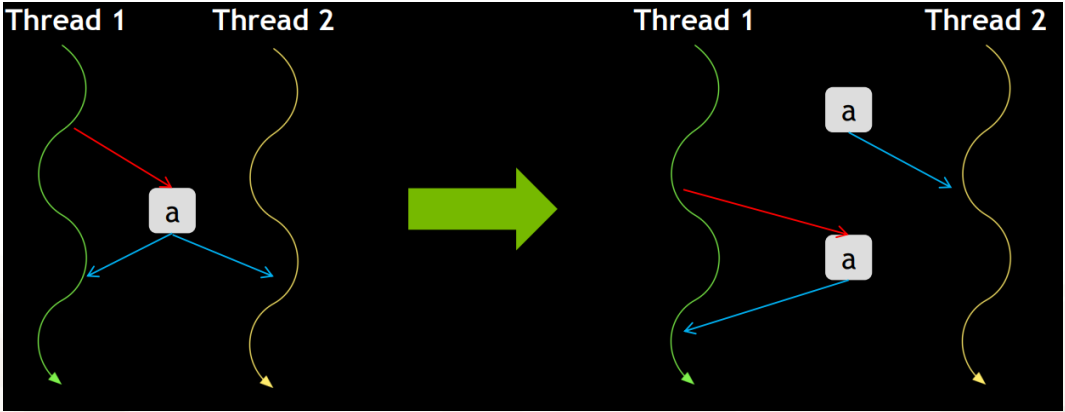
- outil de debug de la mémoire
- ne nécessite pas de recompilation : `cuda-memcheck ./main`
- Permet de faciliter la détection de :
  - fuites mémoire
  - erreurs mémoire (out of bounds, accès non aligné, illegal instruction, ...)
  - race conditions (situations de concurrence)
  - mémoire lue et non initialisée
  - barrières mal placée
- Afficher la ligne où se trouve l'erreur :
  `-Xcompiler -rdynamic -lineinfo`

cuda-memcheck : exemple *race condition*

```
__global__ int bcast(void) {
    int x;
    __shared__ int a;
    if (threadIdx.x == WRITER)
        a = threadIdx.x;
    x = a;
    // do some work
}
```



cuda-memcheck : exemple *race condition*



Où se situe le problème ? (spoiler solution : `__syncthreads()`)

```
cuda-memcheck --tool=racecheck --racecheck-report analysis
```

Sommaire

1 Rappels

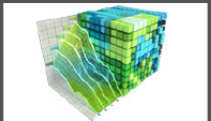
2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

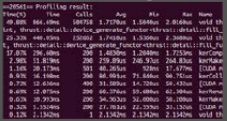
Outils de profiling en CUDA

NSIGHT

NVVP



NVPROF



NVIDIA Provided

TAU

VampirTrace

3rd Party

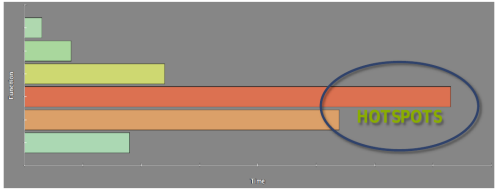
profiling : nvprof

nvprof, profiling en ligne de commande :

- pendant de gprof pour CUDA
- calcul du temps d'exécution de chaque noyau
- calcul des temps de transfert mémoire entre host et device
- collecte des métriques et des événements
- peut suivre des hiérarchies complexes de processus
- ne nécessite pas de recompilation
- peut créer un fichier de données pour NVPP (Nvidia Visual Profiler)

Profiling efficace

Comme toujours, on se focalise sur les *points chauds* ie les optimisations qui apporteront le plus de bénéfice :



L'optimisation peut intervenir n'importe où :

Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

Exemple canonique : addition de vecteurs (côté device)

Reprise du code de la somme de deux vecteurs $C = A + B$: chaque thread effectue une addition de paires

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i < n)
    C[i] = A[i] + B[i];
}
```

Exemple canonique : addition de vecteurs (côté host)

Méthode 1 : la fonction `ceil()` nous assure du bon arrondi

```
void vecAdd(float* hA, float* hB, float hC, int n)
{
  // dA, dB, dC deja alloues et les donnees A et B deja copiees
  vecAddKernel<<< ceil(n/256.0) , 256 >>>(dA, dB, dC, n);
}
```

Méthode 2 :

```
void vecAdd(float* hA, float* hB, float hC, int n)
{
  dim3 dimGrid( (n-1)/256 + 1, 1, 1 );
  dim3 dimBlock(256, 1, 1);
  vecAddKernel<<< dimGrid, dimBlock >>>(dA, dB, dC, n);
}
```

Rappel : pourquoi on doit tester $i < n$ dans le code device ?

Déclaration de fonctions CUDA

	s'exécute sur	s'appelle depuis
`__device__` float DeviceFunc()	device	device
`__global__` void KernelFunc()	device	host
`__host__` float HostFunc()	host	host

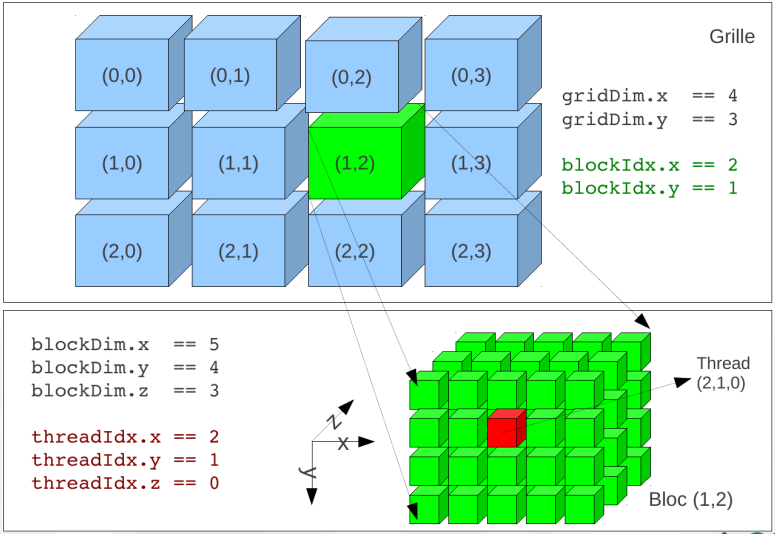
- `__global__` définit un noyau, doit forcément retourner `void`
- `__device__` et `__host__` peuvent être utilisés ensemble
- `__host__` est optionnel si utilisé seul

Types CUDA

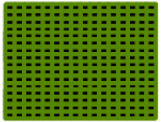
- Types vectoriels : {char, short, uint, int, float, long, ulong}{1-4}
  Exemple : `float4 data[2]; data[1].y = 0.1f;`
- Fonctions mathématiques :
  - sqrtf(), sinf(), log10f(), ...
  - versions rapides : __sinf(), __log10f(), pas de __sqrtf()

Modificateur	Déclaration	Mémoire	Scope	Lifetime
`__device__ __local__`	`int localVar;`	local	thread	thread
`__device__ __shared__`	`int sharedVar;`	shared	block	block
`__device__`	`int globalVar;`	global	grid	application
`__device__ __constant__`	`int constantVar;`	constant	grid	application

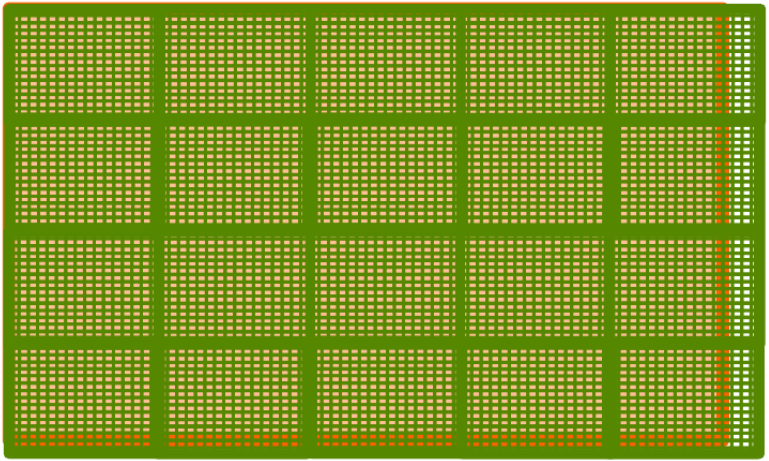
Rappel : organisation des threads



Exemple : parcours d'une image avec grille 2D

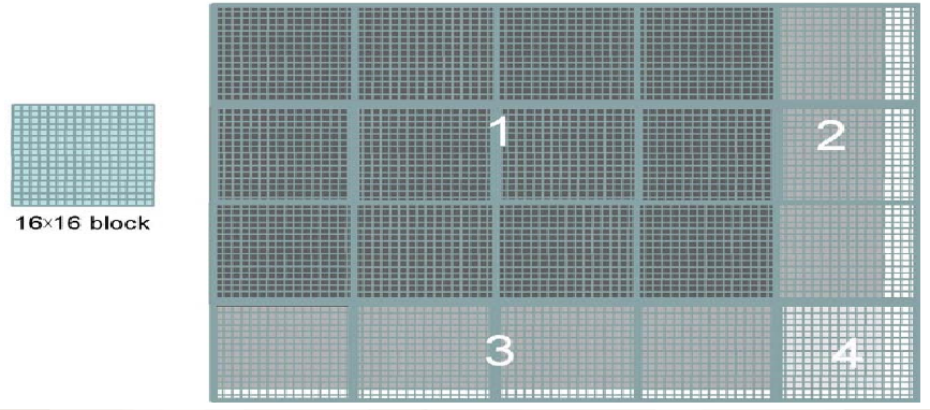


16×16 blocks



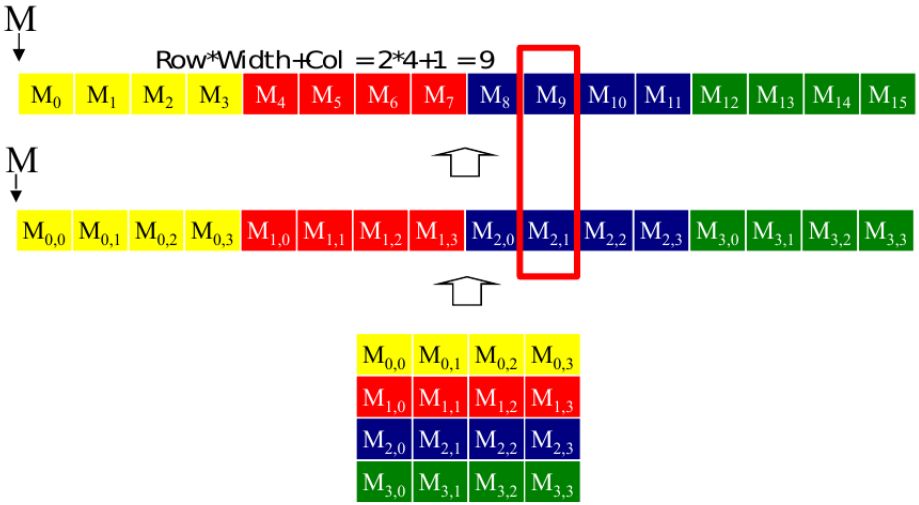
62×76 picture

Exemple : parcours d'une image avec grille 2D



. . .donc tous les threads d'un bloc ne prendront pas le même chemin d'exécution dans le flux de contrôle.

Rappel : passage d'une image 2D à un bloc mémoire plat 1D



Exemple : parcours d'une image avec grille 2D

Kernel côté device :

```
__global__ void PictureKernel(float* dPin, float* dPout, int height, int width)
{
  // Compute row and column number of dPin and dPout element
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  // Each thread computes one element of dPout if in range
  if(row < height && col < width)
  {
    dPout[row*width+col] = 2.0 * dPin[row*width+col];
  }
}

```

Exemple : parcours d'une image avec grille 2D

Code host pour appeler le kernel :

```
// Assume that :
// - the picture is m * n
// - m pixels in y dimension
// - n pixels in x dimension
// - input dPin has been allocated on and copied to device
// - output dPout has been allocated on device
...
dim3 dimGrid( (n-1)/16 + 1, (m-1)/16 + 1, 1);
dim3 dimBlock(16, 16, 1);
PictureKernel<<< dimGrid, dimBlock >>>(dPin, dPout, m, n);
...

```

Exemple avec mémoire partagée

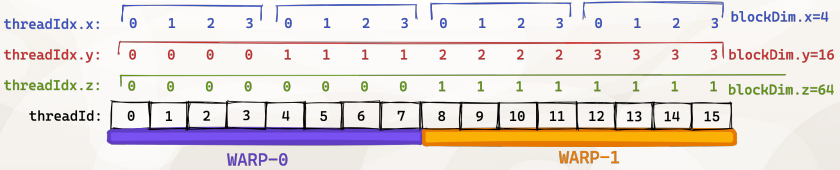
Kernel côté device :

```
__global__ void PictureKernel(float* dPin, float* dPout, int height, int width)
{
  __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int index_in = row * width + col;

  if(row < height && col < width)
  {
    // copie dans la memoire partagee
    block[threadIdx.y][threadIdx.x] = dPin[index_in];
  }
  __syncthreads();

  unsigned int index_out = index_in;

  // Traitement, utilisant la memoire partagee
  dPout[index_out] = 2.0 * block[threadIdx.y][threadIdx.x];
}

```

Mémoire partagée : accès optimal

Revenons sur le concept de *warp* :

- pour l'execution, les threads d'un block sont groupés en *warp*
- un *warp* == 32 threads
- ensuite un warp est affecté à un *warp scheduler* (coeur physique exécutant les instructions)
- chaque multiprocessor possède 4 *warp scheduler*
- le groupement des threads dans chaque warp se fait par *threadId* consécutifs
- par exemple avec un *blockDim* multi-dimensions, on a :
  `threadId = threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*threadIdx.z)`
- alors les thread avec des id contigus font partie du même warp
- Exemple en imaginant un warp à 8 threads (pour de vrai c'est 32) :

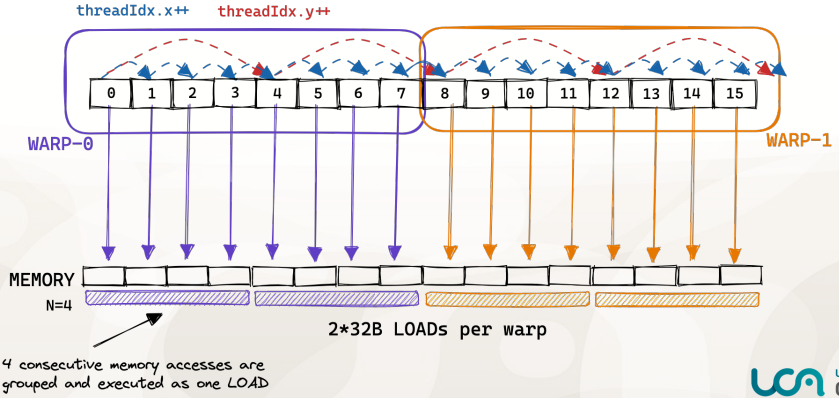


`threadId = threadIdx.x + blockDim.x*threadIdx.y + blockDim.x*blockDim.y*threadIdx.z`

Mémoire partagée : accès optimal via les warp (1)

Accès mémoire par les threads d'un warp :

- les accès mémoire des threads d'un même warp peuvent être groupés
- ces accès peuvent être exécutés comme un seul accès
- Nvidia appelle cela *global memory coalescing*
- Exemple : chaque warp exécute 8 accès mémoire nécessitant seulement 2 chargements de 32 octets



Mémoire partagée : accès optimal via les warp (2)

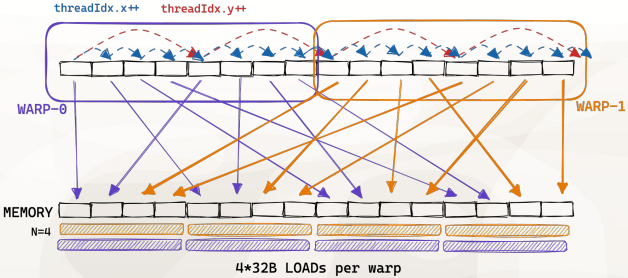
- Le GPU permet des accès de 32, 64 et 128 octets
- Si chaque thread charge un float de 32 bits depuis la mémoire globale
- le warp scheduler peut grouper à 32 * 4 octets = 128 octets en une seule transaction
- Possible seulement si les floats à lire sont contigus en mémoire
- et si les thread id sont consécutifs
- Sinon, il y aura autant d'accès de 32 bits que nécessaire, non groupés
- avec une forte perte de bande passante

Mémoire partagée : accès optimal via les warp (3)

Exemple où ça se passe mal :

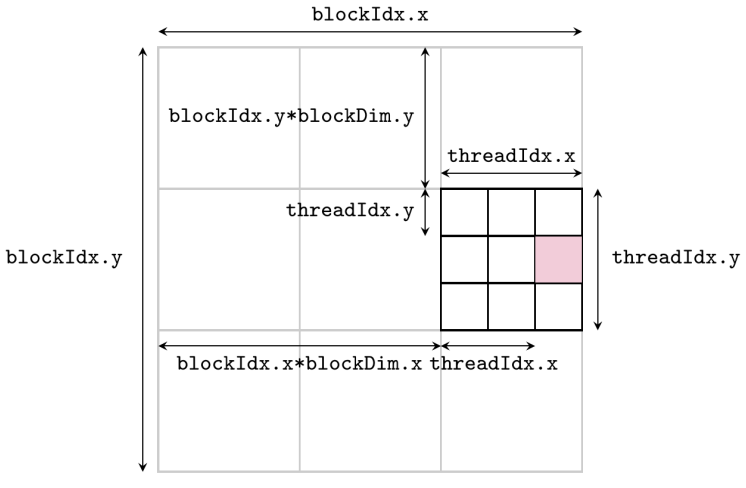
```
const uint x = blockIdx.x * blockDim.x + threadIdx.x;
const uint y = blockIdx.y * blockDim.y + threadIdx.y;
...
float tmp = 0.0;
for (int i = 0; i < K; ++i)
  tmp += A[x * K + i] * B[i * N + y];

C[x * N + y] = alpha * tmp + beta * C[x * N + y];
```

Les accès dans A sont non consécutifs en mémoire en fonction du thread id.



Des blocks et des threads (1)

https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf



Des blocks et des threads (2)

Mais au fait, pourquoi s'embêter avec des threads et des blocks ?

- 1 thread = 1 coeur
- 1 block = 1 SM
- 1 block est exécuté sur 1 SM
- Blocks :
  - Les blocks sont exécutés dans n'importe quel ordre, séquentiellement ou en parallèle
  - Avantage : passage à l'échelle automatique avec le nombre de SM
- Contrairement aux blocks, les threads peuvent
  - Communiquer
  - Se synchroniser
- Ces opérations sont **à l'intérieur d'un bloc**

Des blocks et des threads (3)

Paramètres du kernel :

- Les blocks :
  - Pour que tous les SM travaillent, le nombre de blocks doit être supérieur au nombre de SM
  - Le mieux est d'avoir plusieurs blocks par SM, afin que des blocks s'exécutent pendant une synchro
    $\hookrightarrow$ Donc si synchro utilisée, il vaut mieux utiliser des petits blocks
- Les threads :
  - Les SM ordonnancent les threads par groupe SIMD de 32 (un *warp*). Les threads d'un warp sont synchronisés
    $\hookrightarrow$ Les threads dans un block sont exécutés par groupe de 32.
  - Un SM peut exécuter plusieurs blocks de manière concurrente
  - Optimisation : utiliser des blocks de taille multiple de la taille du warp

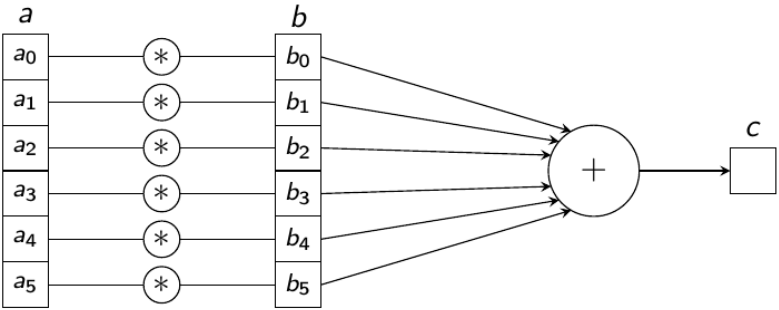
Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

Produit scalaire (dot product)



$$c = (a_0, a_1, a_2, a_3, a_4, a_5).(b_0, b_1, b_2, b_3, b_4, b_5)$$
$$= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + a_5 b_5$$

Produit scalaire : 1 seul block

```
__global__ void dot(int *a, int *b, int *c)
{
  // chaque thread calcule le produit d'une paire
  int temp = a[threadIdx.x] * b[threadIdx.x];
}
```

- Le calcul est local au processus
- Les variables `temp` ne sont pas accessibles aux autres processus
- Mais il faut partager les données pour faire la somme finale

Partager les données entre les threads d'un même block

Rappels :

- Les threads d'un block partagent une zone mémoire appelée **shared memory**
- Caractéristiques :
  - Extrêmement rapide (accès mémoire globale : 400 cycles, accès registre ou mémoire partagée : 1 cycle)
  - *on-chip*
  - Déclarée avec `__shared__`
- Sur le même SM, des blocks partagent la même shared memory.
  ↪ Donc pour une shared memory de 64Ko avec N blocks sur le même SM, chaque block possédera 64/N Ko de mémoire partagée...

Produit scalaire : 1 seul block

```
1  #define N 512 // taille du tableau
2
3  __global__ void dot(int *a, int *b, int *c)
4  {
5    __shared__ int temp[N];
6
7    // chaque thread calcule le produit d'une paire
8    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
9
10   // le thread 0 effectue la somme
11   if(0 == threadIdx.x)
12   {
13     int sum = 0;
14     for(int i = 0; i < N; i++)
15         sum = sum + temp[i];
16     *c = sum;
17   }
18 }
19
```

Que manque-t-il ?

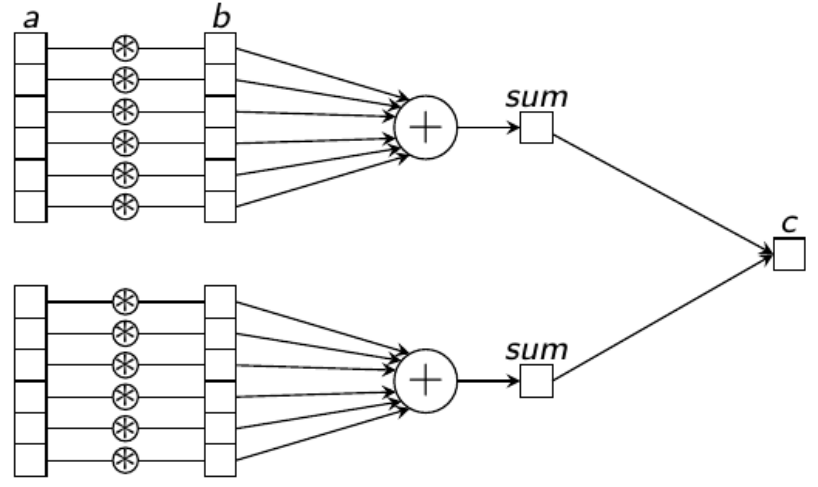
Produit scalaire : 1 seul block

```
1  #define N 512 // taille du tableau
2
3  __global__ void dot(int *a, int *b, int *c)
4  {
5    __shared__ int temp[N];
6
7    // chaque thread calcule le produit d'une paire
8    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
9
10   // synchro pour etre sur que tous les threads ont fini
11   __syncthreads();
12
13   // le thread 0 effectue la somme
14   if(0 == threadIdx.x)
15   {
16     int sum = 0;
17     for(int i = 0; i < N; i++)
18         sum = sum + temp[i];
19     *c = sum;
20   }
21 }
22
```

Produit scalaire : 1 seul block - le main

```
1  int main(void)
2  {
3    int *a, *b, c;
4    int *gpu_a, *gpu_b, *gpu_c;
5    int size = N * sizeof(int);
6    cudaMalloc((void**)&gpu_a, size);
7    cudaMalloc((void**)&gpu_b, size);
8    cudaMalloc((void**)&gpu_c, sizeof(int));
9    a=(int*)malloc(size);
10   b=(int*)malloc(size);
11
12   random_ints(a, N);
13   random_ints(b, N);
14
15   cudaMemcpy(gpu_a, a, size, cudaMemcpyHostToDevice);
16   cudaMemcpy(gpu_b, b, size, cudaMemcpyHostToDevice);
17
18   dot<<<1, N>>>(gpu_a, gpu_b, gpu_c);
19
20   cudaMemcpy(&c, gpu_c, sizeof(int), cudaMemcpyDeviceToHost);
21
22   free(a);  free(b);
23   cudaFree(gpu_a);  cudaFree(gpu_b);  cudaFree(gpu_c);
24   return 0;
25 }
26
```

Plus de parallélisme : plusieurs blocks



Produit scalaire : 1 seul block

```
1  #define N (2048*2048)
2  #define THREADS_PER_BLOCK 512
3
4  __global__ void dot(int *a, int *b, int *c)
5  {
6    __shared__ int temp[THREADS_PER_BLOCK];
7
8    // chaque thread calcule le produit d'une paire
9    int indice = threadIdx.x + blockIdx.x * blockDim.x
10   temp[indice] = a[indice] * b[indice];
11
12   // synchro dans le block
13   __syncthreads();
14
15   // le thread 0 effectue la somme
16   if(0 == threadIdx.x)
17   {
18     int sum = 0;
19     for(int i = 0; i < THREADS_PER_BLOCK; i++)
20         sum = sum + temp[i];
21     *c = sum;
22   }
23 }
24
```

Race condition

- c est dans la mémoire globale
- chaque block a un thread numéro 0, donc plusieurs accès en même temps
- Les opérations atomiques :
  - opérations de lecture, modification, écriture : ininterruptibles
  - plusieurs opérations atomiques possibles :

- `atomicAdd()`
- `atomicSub()`
- `atomicMin()`
- `atomicMax()`
- `atomicInc()`
- `atomicDec()`
- `atomicExch()`
- `atomicCAS()`

Produit scalaire : 1 seul block

```
1  #define N (2048*2048)
2  #define THREADS_PER_BLOCK 512
3
4  __global__ void dot(int *a, int *b, int *c)
5  {
6    __shared__ int temp[THREADS_PER_BLOCK];
7
8    // chaque thread calcule le produit d'une paire
9    int indice = threadIdx.x + blockIdx.x * blockDim.x
10   temp[indice] = a[indice] * b[indice];
11
12   // synchro dans le block
13   __syncthreads();
14
15   // le thread 0 effectue la somme
16   if(0 == threadIdx.x)
17   {
18     int sum = 0;
19     for(int i = 0; i < THREADS_PER_BLOCK; i++)
20         sum = sum + temp[i];
21     atomicAdd(*c, sum);
22   }
23 }
24
```

Produit scalaire : plein de blocks - le main

```
int main(void)
{
  int *a, *b, c;
  int *gpu_a, *gpu_b, *gpu_c;
  int size = N * sizeof(int);
  cudaMalloc((void**)&gpu_a, size);
  cudaMalloc((void**)&gpu_b, size);
  cudaMalloc((void**)&gpu_c, sizeof(int));
  a=(int*)malloc(size);
  b=(int*)malloc(size);

  random_ints(a, N);
  random_ints(b, N);

  cudaMemcpy(gpu_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_b, b, size, cudaMemcpyHostToDevice);

  dot<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(gpu_a, gpu_b, gpu_c);

  cudaMemcpy(&c, gpu_c, sizeof(int), cudaMemcpyDeviceToHost);

  free(a);  free(b);
  cudaFree(gpu_a);  cudaFree(gpu_b);  cudaFree(gpu_c);
  return 0;
}
```

Retour sur les performances d'un GPU

- Tous les threads accèdent à la mémoire globale pour lire les éléments en entrée
  - Un accès mémoire (4 octets) par addition de float
  - 4B/s de bande passante mémoire par FLOP
- En supposant un GPU avec
  - Un pic de calcul flottant (FLOPS) de 1500 GFLOPS avec une bande passante de DRAM de 200GB/s
  - $4 \times 1500 = 6000 GB/s$ nécessaires pour atteindre le pic de FLOPS
  - La bande passante mémoire de 200GB/s limite donc l'exécution à 50 GFLOPS !
- Cette limite permet donc d'atteindre uniquement un pic d'exécution à 3,3% (50/1500) de la capacité théorique de la carte !
- Il est donc nécessaire de couper drastiquement la quantité d'accès mémoire pour se rapprocher de la vitesse théorique de 1500 GFLOPS
- ou de densifier les calculs

Retour sur les performances d'un GPU

Le CPU :

- transfert CPU / RAM à 8 Go/s, 1 float 32 bits = 4 octets $\Rightarrow 2 \times 10^9$ float transférés/s
- Calcul CPU : SSE float 32 bits 4 opérations par cycle @ 3 Ghz $\Rightarrow 4 \times 3 = 12 \times 10^9$ float traités par seconde = 12 GFLOPS, 4 coeurs $\Rightarrow$ 48 GFLOPS

$\Rightarrow$ appli optimale : au minimum 24 opérations par float transféré

Le GPU :

- Débit mémoire vs débit instructions
  - transfert GPU / G-RAM à 115Go/s
  - 1 float 32 bits donc 29×10^9 float transférés/s
  - calcul GPU : 994 GFLOPS

$\Rightarrow$ appli optimale : au minimum 34 opérations/float transféré
Mais si transfert GPU / RAM PC à 5Go/s, alors appli optimale à 775 opérations/float transféré.

Outil : CUDA calculator

Pour aider à choisir la configuration optimale : `https://xmartlabs.github.io/cuda-calculator/`

CUDA Occupancy Calculator

Compute Capability version	5.0
Threads per block	128
Registers per thread	8
Shared memory per block	1024

Calculate

GPU Occupancy Data is displayed here and in the graphs

Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	16
Occupancy of each Multiprocessor	1

Physical Limits for GPU Compute Capability

Version	5.0
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	32
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory per Multiprocessor (bytes)	65536
Shared Memory Allocation unit size	256
Warp allocation granularity (for register allocation)	4

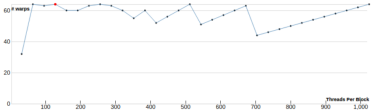
Allocation Per Thread Block

Warps	4
Registers	1024
Shared Memory	1024

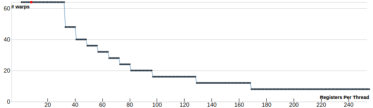
Maximum Thread Blocks Per Multiprocessor

Limited by Max Warps / Blocks per Multiprocessor	16
Limited by Registers per Multiprocessor	64
Limited by Shared Memory per Multiprocessor	64

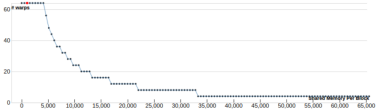
Impact of Varying Block Size



Impact of Varying Register Count Per Thread



Impact of Varying Shared Memory Usage Per Block



Optimisations du code CUDA

- Vitesse GPU / Vitesse mémoire globale
  - Dérouler les boucles ⇒ davantage de traitement *utile* par thread (par rapport au nombre d'instructions nécessaires pour maintenir la boucle)
  - Davantage de threads par block ⇒ cacher la latence de la mémoire globale
  - Limitation par la taille du jeu de registres
- Utiliser la mémoire partagée
  - Aussi rapide que les registres ≈ 1 cycle vs 400 cycles pour la mémoire globale
  - Adapter les algorithmes pour partager les données
  - Limitation par la taille de la mémoire partagée
- Limiter la divergence des threads
  - Utiliser les `if` avec parcimonie

Optimiser les algorithmes pour le GPU

- Maximiser le parallélisme indépendant
- Maximiser l'intensité arithmétique
  ↪ cf les TP, augmentation de la quantité de calculs par rapport à la bande passante
- Parfois, il est plus efficace de recalculer que de mettre en cache :
  - le GPU dépense ses transistors dans les ALU, pas dans la mémoire
  - le nombre de registres est limité par block
  - la mémoire partagée est limitée
- Faire plus de calculs sur le GPU pour éviter les transferts de données coûteux
  ↪ Même du calcul faiblement parallélisé peut parfois être plus rapide que transférer et rapatrier entre host et device
  ↪ Quitte à transférer, privilégier un gros transfert plutôt que de nombreux petits (pénialité de synchronisation)

Exemple d'optimisation

Exemple d'optimisation sur un kernel de produit scalaire parallèle (réduction) :
http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 16M elements: 72 GB/s!

Optimiser les algorithmes pour le GPU

- Utiliser l'efficacité du parallélisme :
  - Partitionner les calculs de manière à maintenir les *streaming multiprocessors* sur GPU équitablement occupés
    $\hookrightarrow$ plein de thread, plein de blocks de threads
  - Utiliser les ressources critiques avec parcimonie pour s'assurer que plusieurs blocks de threads soient actifs sur chaque multiprocesseur
    $\hookrightarrow$ les registres et la mémoire partagée sont des ressources critiques
- Comment trouver la taille des blocks et de la grille ? Heuristiques :
  - # de blocks / # de SM > 1
    $\Rightarrow$ Ainsi chaque SM a au moins 1 block à exécuter
  - Au max la moitié des ressources partagées entre les blocks
    $\Rightarrow$ mémoire partagée, registres
    $\Rightarrow$ plusieurs blocks peuvent tourner en même temps sur un SM
    $\Rightarrow$ si plusieurs blocks n'attendant pas sur un `__syncthreads` coexistent, alors le GPU peut rester occupé
  - # de blocks / # de SM > 2
    $\Rightarrow$ plusieurs blocks peuvent tourner en même temps sur un SM
  - # de blocks $> 100 \Rightarrow$ pour passer à l'échelle sur les futurs GPU
    $\Rightarrow$ 1000 blocks permettent déjà de passer plusieurs générations de GPU

Optimiser les algorithmes pour le GPU

- Rendre son application paramétrable
  - permet de s'adapter à différents GPU
  - les GPU varient grandement : # de SM, bande passante mémoire, taille de la mémoire partagée, taille des registres, nombre de threads par bloc, ...
- Certaines applications sont auto-paramétrables :
  - exemples : FFTW et ATLAS
  - font varier automatiquement les paramètres pour trouver l'optimum du GPU en cours
  - et enregistrent la config optimale
- Le *must* : ne pas faire de code CUDA :
  - utiliser des bibliothèques déjà prêtes utilisant CUDA avec un fort retour d'expérience
  - certaines bibliothèques sont mêmes proposées par Nvidia
  - fournissent un remplacement exact de la bibliothèque initiale
  - exemples : BLAS → cuBLAS, FFTW → cuFFT, version CUDA parallèle de la STL (Thrust), écriture / lecture des JPEG avec nvJPEG, modules d'accélération CUDA dans OpenCV et dans FFMPEG, ...
  - bibliothèque CUDA-parallèle pour les réseaux de neurones : cuDNN

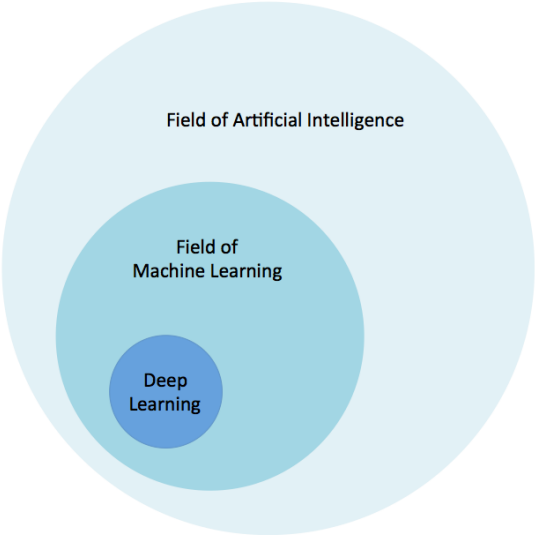
Sommaire

1 Rappels

2 Architecture GPU

3 Programmation GPU
- Introduction CUDA
- Organisation des threads en CUDA
- La mémoire en CUDA
- Les threads sur le GPU
- Debug CUDA
- Profiling CUDA
- Programmer en CUDA
- Exemple et analyse mémoire partagée : produit scalaire
- Deep learning et CUDA

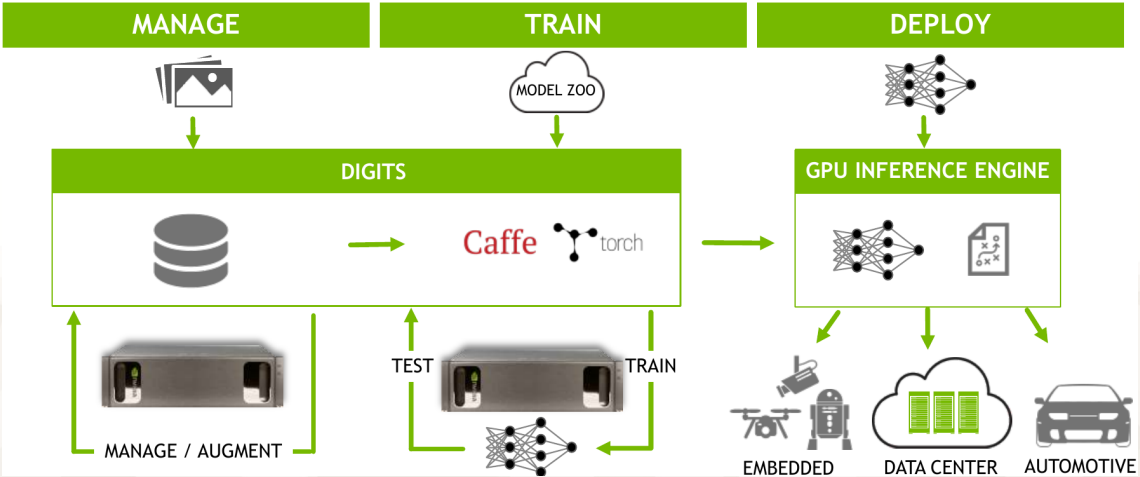
Hiérarchie


Field of Artificial Intelligence
Field of
Machine Learning
Deep
Learning

Définitions

Intelligence artificielle : l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence.
Machine learning (apprentissage automatique) : un champ d'étude de l'intelligence artificielle qui se fonde sur des approches mathématiques et statistiques pour donner aux ordinateurs la capacité d' « apprendre » à partir de données, c'est-à-dire d'améliorer leurs performances à résoudre des tâches sans être explicitement programmés pour chacune. Plus largement, il concerne la conception, l'analyse, l'optimisation, le développement et l'implémentation de telles méthodes.
Version courte : discipline de l'informatique qui donne aux ordinateurs la capacité d'apprendre sans être explicitement programmé.
Réseau de neurones artificiels : un algorithme d'apprentissage basé sur les réseaux de neurones artificiels, habituellement appelé un réseau de neurones, est un algorithme d'apprentissage dont le fonctionnement est inspiré par la structure et les aspects fonctionnels des réseaux de neurones biologiques.
Deep learning (apprentissage profond) : consiste en plusieurs niveaux cachés dans un réseau de neurones artificiels.

Workflow habituel



Exemples d'utilisation

Où utilise-t-on du Deep Learning ?

- Météo : prédictions plus rapides qu'avec un modèle classique de dynamique des fluides
- Jeux vidéos : DeepMind (IA Starcraft 2), Dota 2 OpenAI, Bots LoL, ...
- Bourse : prédictions des actions, des tendances, prise de décision extrêmement rapide
- Rendu 3D (jeux vidéos) : meilleure résolution à coût moindre
- Retouche et amélioration d'images, super résolution, reconstruction de parties manquantes, reconstruction 3D
- Deep fake vidéo et audio (synthèse de la parole, synthèse d'images)
- Génération automatique de textes (articles scientifiques ...), traduction
- Véhicules autonomes, robotique : reconnaissance des panneaux, prise de décision rapide, analyse des quantités de données générées par les capteurs, ...
- Caméras intelligentes : comportements, détection de zone d'intérêt, identification, ...
- Santé : détection de maladies sur de l'imagerie (IBM Watson), analyse temps réel de signaux (ECG)
- Art : Google DeepDream, musique (composition automatique), peinture, ...

Exemple traitement d'images

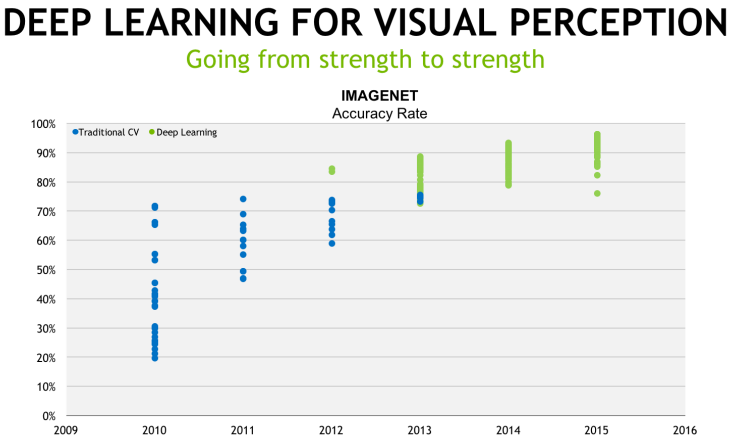
Références :

- Google DeepDream : https://deepdreamgenerator.com & https://github.com/google/deepdream
- Reconstruction 3D : https://blogs.nvidia.com/blog/2019/12/09/neurips-research-3d/
- Upscaling, super résolution dans les jeux vidéos : https://www.nvidia.com/en-us/geforce/news/dlss-control-and-beyond/
- Composition de musique automatique : https://www.digitaltrends.com/mobile/huawei-ai-unfinished-symphony/ & https://bit.ly/2sF801e
- Peinture (painting style transfer) : https://deepart.io
- Santé : https://www.wired.co.uk/article/ibm-watson-medical-doctor
- Génération automatique d'articles scientifiques : http://news.mit.edu/2015/how-three-mit-students-fooled-scientific-journals-0414
- Bourse (High Frequency Trading) : https://towardsdatascience.com/neural-networks-deep-learning-the-revival-of-hft-2bc2c271fba2 & https://arxiv.org/pdf/1809.01506.pdf

GUI interactive pour jouer avec des réseaux de neurones simples : https://playground.tensorflow.org

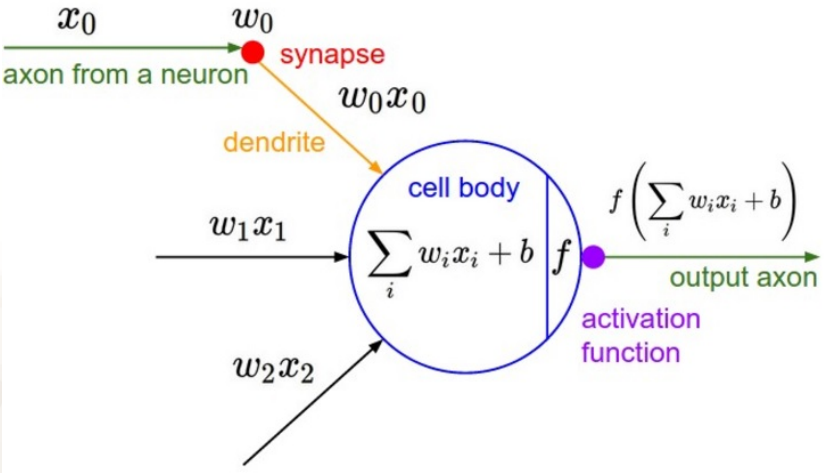
Exemple traitement d'images

Précision progressive des résultats de la compétition ImageNet montrant que depuis 2012, les solutions à base de Deep Learning dominent les tâches de classification de 10 millions d'images.



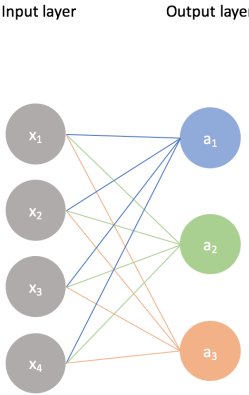
Pourquoi les GPU sont bons en DL ?

Représentation du neurone artificiel :



Pourquoi les GPU sont bons en DL ?

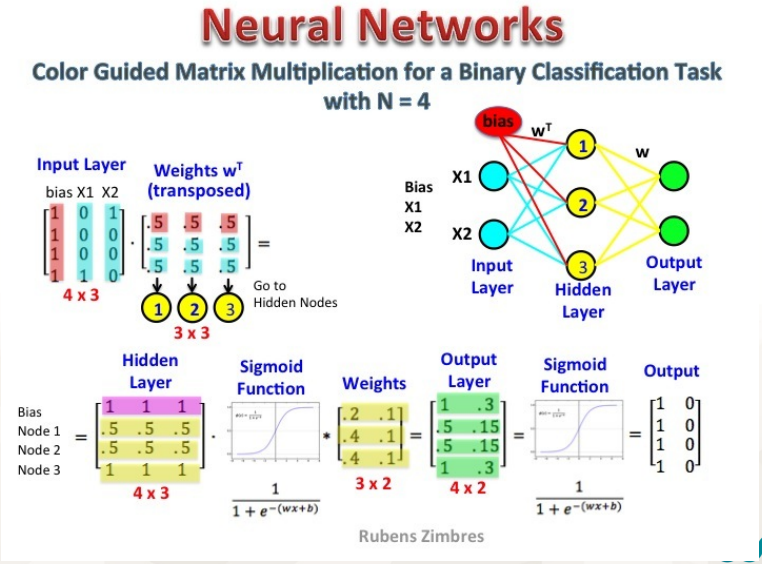
Représentation matricielle du réseau de neurones :



A simple neural network

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow[activation]{} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Pourquoi les GPU sont bons en DL ?



Pourquoi les GPU sont bons en DL ?

- DL : ce sont des mathématiques
- NN : intrinsèquement parallèles
- Grande quantité d'opérations matricielles
- Beaucoup de calculs sur des float, grande densité de calculs

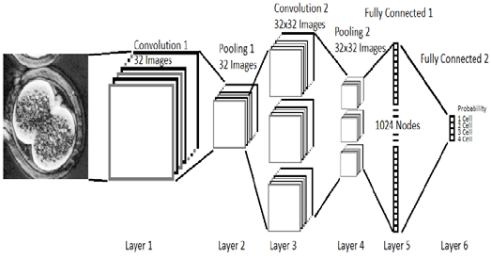
Le GPU fournit :

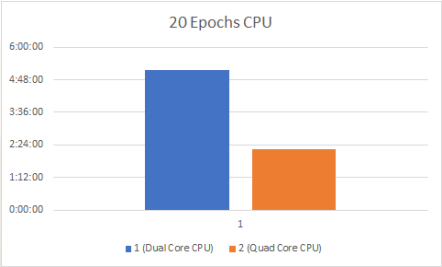
- de la précision dans la prédiction
- des résultats rapides
- des empreintes plus faibles
- une consommation moindre

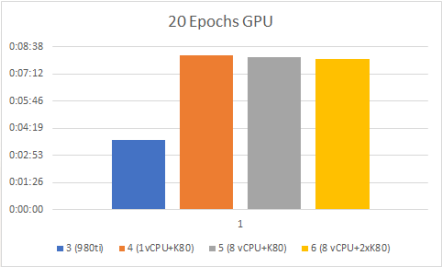
Pourquoi les GPU sont bons en DL ?

Les GPU sont bons en DL

Exemple : images en entrée 320x320, réseau possédant 16832432 paramètres à entraîner (`http://csis.pace.edu/~ctappert/srd2017/2017PDF/c1.pdf`)







Bibliothèques GPU pour Deep Learning

Plusieurs bibliothèques disponibles proposant l'accélération GPU (CUDA, OpenCL) :
`https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software`

- Keras
- PyTorch
- Torch
- Theano
- TensorFlow

Qu'en est-il du monde Deep Learning hors plates-bandes fortifiées de CUDA ?
`towardsdatascience.com/on-the-state-of-deep-learning-outside-of-cudas-walled-garden-d88c8bbb4342`