

# Introduction programmation GPU PyCuda

21 février 2020

# Programmation sous GPU ?

- Matériel
  - Utilisation du GPU (au lieu du CPU) pour réaliser des calculs
  - CPU : Central Processing Unit (processeur)
  - GPU : Graphical Processing Unit (processeur dans les cartes graphiques)
  - Un moyen pour programmer en parallélisant les codes
- Langages de programmation
  - Pour réaliser un prg qui tourne en utilisant le CPU : C/C++, fortran, python,...
  - Pour réaliser un prg qui tourne en utilisant le GPU : OpenCL, CUDA C/C++
  - CUDA associé à la marque *NVidia*
  - CUDA mixe du code C/C++ sur CPU avec du code 'CUDA' (une sorte de C aussi) sur le GPU.  
Concrètement : un code C/C++ usuel qui délègue une partie des calculs sur le GPU

- Dans ce cours : utilisation du GPU interfacée avec python  
Concrètement : un code python qui délègue une partie des calculs sur le GPU  
Permet d'avoir un langage de haut niveau (avec bcp de 'library', stat, graphe, numérique,...) mais lent et un langage de bas niveau sur le GPU (type C) mais rapide.

### But du calcul sur GPU :

- Obtenir des calculs numériques plus rapides en parallélisant les codes
- Il faut que les calculs se découpe en morceaux indépendants : Monte Carlo, méthodes numériques en analyse (EDP, EDO), ...

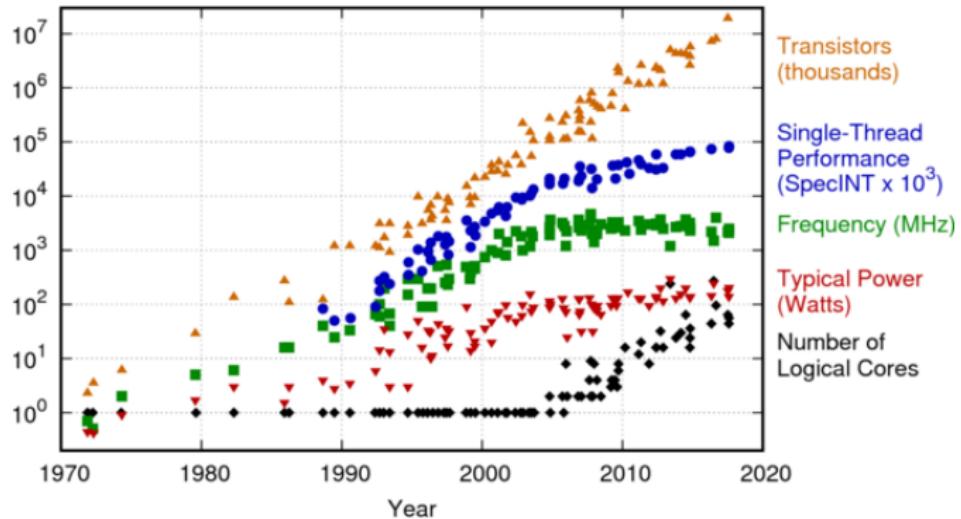
# Puissances de calcul actuelles (et comparaison)

- CPU haut de gamme 2018 :
  - intel i7 9700k (460€) 8 cœurs × 2 unités de calculs par cœurs × unités type 'vec 8' → 128 calculs possibles en même temps
  - cadencé à 3.6Ghz → 460 gflops (double si FMA)
  - Code C non parallèle : quelques gflops seulement...  
Exploiter CPU en parallèle (C avec OpenMP, OpenCL,...)
- GPU en 2018 :
  - GPU GTX 1050 (170€) : 768 unités de calculs  
@1.5Ghz → 1152 gflops (double si FMA) > 1 teraflops !
  - GPU RTX 2080 (700€), 2944 unités de calculs et > 10 teraflops

# Puissances de calcul actuelles (et comparaison)

- CPU haut de gamme 2018 :
  - intel i7 9700k (460€) 8 cœurs x 2 unités de calculs par cœurs x unités type 'vec 8' → 128 calculs possibles en même temps
  - cadencé à 3.6Ghz → 460 gflops (double si FMA)
  - Code C non parallèle : quelques gflops seulement...  
Exploiter CPU en parallèle (C avec OpenMP, OpenCL,...)
- GPU en 2018 :
  - GPU GTX 1050 (170€) : 768 unités de calculs  
@1.5Ghz → 1152 gflops (double si FMA) > 1 teraflops !
  - GPU RTX 2080 (700€), 2944 unités de calculs et > 10 teraflops
- Ordinateur le plus puissant du monde 1997-2000 et premier à dépasser 1 teraflops :  
ASCI Red [https://en.wikipedia.org/wiki/ASCI\\_Red](https://en.wikipedia.org/wiki/ASCI_Red)
- Ordinateur le plus puissant du monde (fin 2018) :  
IBM Séquoia [https://en.wikipedia.org/wiki/IBM\\_Sequoia](https://en.wikipedia.org/wiki/IBM_Sequoia)  
 $1,5 * 10^6$  unités de calculs et > 20000 teraflops

## 42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

[www.karlrupp.net/2018/02/](http://www.karlrupp.net/2018/02/)

42-years-of-microprocessor-trend-data/

- Les fréquences des processeurs n'augmentent plus bcp depuis 15 ans.

## Conclusion

- Puissance de calcul > 1Tflops sur GPU
- Nécessite calcul en parallèle sur plusieurs milliers d'unités de calculs → diviser l'exécution d'un algorithme en des milliers de calculs parallèles
- Codes séquentielles limités à quelques gflops

# Que va-t-on faire ?

Apprendre à programmer sous Cuda au travers de quelques exemples de code :

- ① Manipuler des tableaux de nombres, faire des opérations simples sur des vecteurs / matrices
- ② Dessiner l'ensemble de Julia
- ③ Simuler des v.a. par méthode de rejet et estimer la fonction de répartition de la loi par méthode de Monte Carlo

Quelques références :

- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- Mark Giles lecture on Cuda  
<https://people.maths.ox.ac.uk/gilesm/cuda/>
- Pycuda web page  
<https://mathematician.de/software/pycuda/>

# Programmation parallèle

Idée :

Remplacer les itérations séquentielles d'une boucle par une exécution simultanée du contenu de la boucle.

- Ex 1 : multiplication d'un vecteur (tableau) de taille  $n$

```
void multi (float * tab, float x, int n) {  
    int i;  
    for (i=0;i<n; i++) { tab[i]=x*tab[i] ; } ; }
```

Se parallélise : on peut multiplier par  $x$  toutes les composantes en même temps

- Ex 2 : Addition de deux vecteurs de taille  $n$

```
void addi (float * a, float * b, float *c, int n) {  
    int i;  
    for (i=0;i<n; i++) { c[i]=a[i]+b[i] ; } ; }
```

Se parallélise immédiatement aussi !

# Programmation parallèle

- Ex 3 : Calcul d'une somme

```
float somme ( float * tab , int n ) {  
    float S=0;  
    int i ;  
    for ( i=0;i<n ; i++ ) { S=S+tab [ i ] ; } ;  
    return (S);  
}
```

A priori se parallélise pas : chaque itération nécessite le résultat de l'itération précédente (en fait en modifiant l'ordre des calculs se parallélise..)

- Ex 4 : Calcul d'une suite récurrente  $u_{n+1} = \sin(u_n)$  :

```
float somme ( float u0 , int n ) {  
    int i ;  
    float un=u0;  
    for ( i=1;i<=n ; i++ ) { un=sin (un) ; } ;  
    return (un);  
}
```

A priori se parallélise pas : chaque itération nécessite le résultat de l'itération précédente

- Ex 5 : calcul de  $u_N$ ,  $N$  grand, avec  $u_n = au_{n-1} + b_{n-1}$ ; où  $a, b_n$   $u_0$  données.  
Parallélisable ??

- Ex 5 : calcul de  $u_N$ ,  $N$  grand, avec  $u_n = au_{n-1} + b_{n-1}$ ; où  $a, b_n$  données.

Parallélisable ??

Oui si on écrit  $u_N = \sum_{k=1}^N a^k b_{N-k} + a^N u_0$

# Notre premier programme Cuda

- Paralléliser l'exemple 2 : addition de deux vecteurs (tableaux de dim 1) de taille  $n$  dont un code C serait

```
void addi (float * a, float * b, float *c, int n) {  
    int i;  
    for (i=0;i<n;i++) { c[i]=a[i]+b[i] ; } ; }
```

# Principes généraux de CUDA

- ① L'ordinateur est considéré divisé en 2 :
  - ① La partie 'Host' (hôte) : qui est le CPU, la mémoire centrale et sur lequel un programme en C si on est en CUDA for C, ou en Python dans notre cas, s'exécute
  - ② La partie 'Device' ('dispositif/outil') : qui est le GPU et la mémoire du GPU et sur lequel s'exécute des 'kernels' qui font les calculs en parallèle
- ② La partie 'Host', donne les instructions pour lancer les calculs sur la partie 'Device/GPU' et s'occupe de transférer les données entre mémoire centrale et mémoire du GPU.

## Ecriture du kernel sur la partie device (le GPU)

- On va additionner les deux vecteurs de taille  $n$  en lançant de nombreux '*threads*' (instances d'un programme) qui s'occuperont chacun d'additionner un seul des indices  $i \in \{0, \dots n - 1\}$  des vecteurs
- Chaque thread exécute **exactement le même code** que l'on appelle un '*kernel*' mais il l'exécute sur des données différentes !
- Le kernel est écrit dans le langage CUDA du GPU qui est du C (avec quelques instructions et mots clés en plus) :

```
--global__ void somme(int n, float * a, float * b,
                      float *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index >= n) return ;
    c[index] = a[index] + b[index];
}
```

Le code de ce kernel est enregistré dans un fichier (vec\_sum.cu)

```
__global__ void somme(int n, float * a, float * b,
float *c ) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
if (index >= n) return ;
c[index] = a[index] + b[index];
}
```

Le mot clé `__global__` dit que la fonction somme est une fonction de type 'kernel' :

- qui s'exécute sur le 'device'
- est appellable depuis le code (python chez nous) du 'host'

Un kernel retourne forcément le type `void`

La ligne `int index = threadIdx.x + blockIdx.x *`  
`blockDim.x;` est celle qui calcul l'indice de la case des tableaux  
sur laquelle le thread va travailler

# Organisation des threads

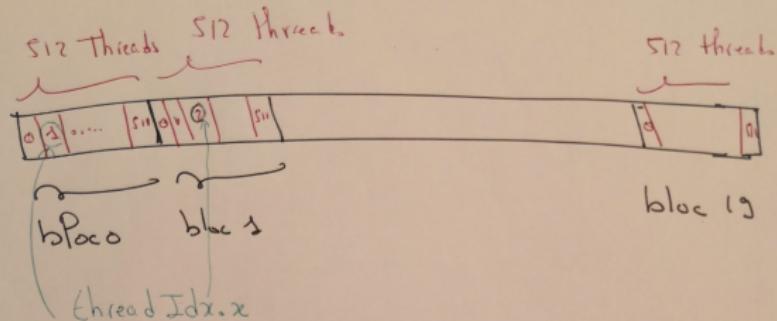
- Pour simplifier supposons que la taille du tableau est  $n = 10000$ . Il faut donc lancer au moins 10000 threads.

On va dire que le kernel doit être lancé sur un grille de thread de dimension 1 (dont on voudrait qu'elle soit de longueur au moins  $n$ ).

Nécessairement sous CUDA les threads sont regroupés par bloc :

- On choisit une taille de block : par exemple `blockDim=512`
- On choisit un nombre de blocks :  $\text{entier}(n/512)+1=20$   
Cela va lancer  $20*512=10240$  threads.
- Chaque thread connaît le numéro du bloc où il est : c'est la variable `blockIdx.x` qui va ici de 0 à 19
- Chaque thread connaît son numéro au sein du bloc où il est : c'est la variable `threadIdx.x` qui va ici de 0 à 511

# La grille de threads en dessin



blockDim.x est le mot clé CUDA pour  
la variable qui contient la taille des blocs  
ici  $\text{blockDim.x} = 512$

On peut calculer l'indice du thread par  
 $\text{index} = \text{blockIdx.x} \times \text{blockDim.x} + \text{Thread Idx.x}$

index va de 0 à 10239 ( $20 \times 512 = 10240$   
Threads)

Pour additionner les deux vecteurs : on lance donc le kernel avec la grille de thread précédent

```
--global__ void somme(int n, float * a, float * b,
                      float *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index >= n) return;
    c[index] = a[index] + b[index];
}
```

- ① Pourquoi la ligne de code `if (index >= n) return;` ?
- ② Ici chaque thread s'occupe d'additionner une seule composante des vecteurs. Autre choix possible : chaque thread additionne deux composantes par exemple (alors nb de thread divisé par 2 et modifier le kernel...)

# Pourquoi regrouper en blocs ?

- ① Les threads de même blocs peuvent communiquer entre eux (utile dans certains algorithmes..)
  - ② Les threads de blocs différents ne peuvent pas communiquer (en fait restriction matérielle)
- 
- ① Le nombre de threads maximum par bloc est 1024 et ce nombre doit être multiple de 32 (limite matérielle cf manuel NVidia CUDA)
  - ② Le nombre max de blocs est  $2^{31} - 1$  (limite matérielle cf manuel NVidia CUDA)

Csq : Le nb de threads lancables simultanément (sur une grille de dim 1) est gigantesque ( $1024 * 2^{31} - 1$ ).

**Question** : combien de threads minimum pour espérer utiliser pleinement le GPU ?

**Question** : combien de threads minimum pour espérer utiliser pleinement le GPU ?

Ici GTX1050 : 768 unités de calcul. Par l'architecture, chaque unité de calcul doit s'occuper de plusieurs threads pour cacher les latences (sur l'architecture AMD 'équivalente' 4 cycles minimum de latency, sur NVidia je n'ai pas vu).

Ordre de grandeur  $4 * 768 \simeq 3000$  est un minimum.

**Question** : combien de threads minimum pour espérer utiliser pleinement le GPU ?

Ici GTX1050 : 768 unités de calcul. Par l'architecture, chaque unité de calcul doit s'occuper de plusieurs threads pour cacher les latences (sur l'architecture AMD 'équivalente' 4 cycles minimum de latency, sur NVidia je n'ai pas vu).

Ordre de grandeur  $4 * 768 \simeq 3000$  est un minimum.

Plus est mieux.

## Déroulement du programme sur le "host" (et le "device")

On veut additionner deux tableaux de nombres  $a$  et  $b$  sous python et stocker dans  $c$

Le déroulé du pgr Python sur l'hôte est le suivant :

- Etape 0 : compiler le kernel 'somme' et le rendre lancable depuis Python. Cela n'a besoin d'être fait qu'une seule fois (au tout début du prg...)
- Etape 1 : Copier les tableaux  $a$  et  $b$  présents sur la mémoire de l'hôte dans deux autres tableaux dans la mémoire GPU ( $a\_GPU$  et  $b\_GPU$ ). Allouer de l'espace pour un tableau  $c\_GPU$  sur la mémoire du GPU pour stocker le résultat de l'addition.
- Etape 2 : Lancer le kernel sur le "device" avec la bonne grille de thread et qui opère le calcul  $c\_GPU = a\_GPU + b\_GPU$ .
- Etape 3 : Copier le tableau  $c\_GPU$  qui contient le résultat dans la mémoire GPU vers le tableau Python  $c$  en mémoire centrale.

# Commandes Pycuda pour allouer de la mémoire GPU

Voir <https://documentarian.de/pycuda/tutorial.html#transferring-data>

- Importer la bibliothèque Python `import pycuda.driver as cuda`

- Allouer un tableau de *nb\_octets* octets sur le GPU :

```
tableau_sur_gpu=cuda.mem_alloc(nb_octets)
```

- Copier un tableau de l'hôte vers le GPU

```
cuda.memcpy_htod(tab_sur_GPU,tab_sur_hote)
```

- Copier un tableau du GPU vers l'hôte

```
cuda.memcpy_dtoh(tab_sur_hote,tab_sur_GPU)
```

Exemple : Creer un tableau qui contient 1000 nombres de type float32 et initialisé à zero.

```
a=np.zeros(1000,dtype=np.float32)
a_GPU=cuda.mem_alloc(a.nbytes)
# ou equivalent ici a_GPU=cuda.mem_alloc(4000)
# car un float32 est code sur 4 octets
cuda.memcpy_htod(a_gpu,a)
```

Rq : le type Python np.float32 est le le même que le type C  
float

# Le code Python en entier

Ouvrir le fichier `vec_sum.py` pour voir le code !

Les imports de library Python

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

Etape 0 : Lire fichier.cu , compiler le code, importer le kernel :

```
f = open("vec_sum.cu", 'r')
cuda_source = "" .join(f.readlines())
mod=SourceModule(cuda_source)
vecteur_somme_kernel=mod.get_function("somme")
```

Attention la dernière ligne qui importe le kernel "somme" doit être répétée pour chaque kernel à importer du fichier (si celui ci a plusieurs kernels)

BLOCKDIM=512

Création de 3 tableaux sous numpy :

```
n=10**4
a=np.arange(0,1,1/n, dtype=np.float32)
b=np.ones_like(a)
c=np.empty_like(a)
```

**Etape 1 : Allocation mem sur le GPU, copie des données depuis l'hôte vers la mem. du GPU**

```
input1_gpu=cuda.mem_alloc(a.nbytes)
input2_gpu=cuda.mem_alloc(a.nbytes)
output_gpu=cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(input1_gpu,a)
cuda.memcpy_htod(input2_gpu,b)
```

**Etape 2 : Lancer le kernel**

```
vecteur_somme_kernel(np.int32(a.size), input1_gpu, input2_gpu,
                      output_gpu, block=(BLOCKDIM,1,1),
                      grid=(a.size//BLOCKDIM+1,1,1))
```

**Etape 3 : Rapatrier le résultat depuis la mémoire du GPU**

```
cuda.memcpy_dtoh(c,output_gpu)
```

## Détail sur le lancement du kernel :

```
vecteur_somme_kernel(np.int32(a.size), input1_gpu,  
                     input2_gpu, output_gpu, block=(BLOCKDIM,1,1),  
                     grid=(a.size//BLOCKDIM+1,1,1))
```

- On lance le kernel par le nom sur lequel il a été importé sous Python (cf etape 0)
- Ensuite on passe les arguments que le kernel attend :

```
__global__ void somme( int taille, float * a, float * b,  
                      float *c )
```

Attention CUDA est explicitement typé comme le C, alors que sous Python les variables n'ont pas de type explicite. Il faut garantir que les variables passées depuis Python ont le type attendu par le kernel.

Correspondance des noms de type NumPy / C :

np.float32 = float

np.float64 = double

np.int32 = int

np.int64 = long long int

PyCuda s'occupe de la correspondance des pointeurs de tableaux

## Détail sur le lancement du kernel :

```
vecteur_somme_kernel(np.int32(a.size), input1_gpu,  
input2_gpu, output_gpu , block=(BLOCKDIM,1,1),  
grid=(a.size//BLOCKDIM+1,1,1))
```

- Les deux derniers **arguments** décrivent la grille de threads qui exécutent le kernel :

ici en dimension 1,  
avec des blocs de taille BLOCKDIM (ici BLOCKDIM=512),  
et un nombre de blocs égale à  $a.size//BLOCKDIM+1$  (c'est à dire 20 ici, car  $a.size=n=10000$ )

Vous pouvez lancer le code et vérifier que  $c = a + b$ .

Pour cela :

- ➊ Se connecter sous Linux
- ➋ Ouvrir une console et taper ... ([voir au tableau](#)) pour que Python trouve le répertoire contenant le compilateur CUDA (nvcc)
- ➌ Lancer spyder sur cette même console et faire tourner les codes pythons dedans

Normalement cela marche !

Rq : l'execution du kernel est instantanée (même avec  $n = 10 ** 6$ ,  $n = 10 ** 7$ ). Le plus long est la compilation du kernel au début.

## Des kernels un peu moins simples

- Le fichier `nom_du_fichier.cu` qui contient le kernel peut contenir plusieurs kernels (alors il faut importer chacun des kernels dans python avec une ligne dans l'étape 0)
- Le fichier `nom_du_fichier.cu` peut aussi contenir des fonctions qui seront exécutées sur le device, peuvent être appelées par le device **mais ne peuvent pas être appelées par l'hôte** (à la différence des kernels).

Ces fonctions sont précédées par le mot clé `__ device __` et elles peuvent retourner une valeur autre que `void`.

```
--device-- float carre (float x);

--global-- void somme_carre(int n, float * a,
                           float * b, float *c ) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index >= n) return ;
    c[index] = a[index] + carre(b[index]);
}

--device-- float carre (float x) {
    return (x*x);
}
```

## A vous de faire

Ecrire les kernels suivants et les lancer depuis Python :

**Exo 1** : Ecrire un kernel qui prend un tableau  $a$  de longueur  $n$  et le "retourne" ( $b[i] = a[n - i - 1]$ ) dans un tableau  $b$  :

```
--global__ void retourne (int n, float * a, float *b)
```

**Exo 2** : Ecrire un kernel qui rempli un tableau  $a$  de longueur  $n$  avec les valeurs  $a[i] = v_{min} + (v_{max} - v_{min}) * (i/n)$  :

```
--global__ void lineaire (int n, float * a,
                           float v_min, float v_max)
```

**Exo 3** : Ecrire un kernel qui rempli un tableau  $a$  de longueur  $n$  avec les valeurs  $a[i] = \cos(v_{min} + (v_{max} - v_{min}) * (i/n))$  :

```
--global__ void tableau_cos (int n, float * a,
                                float v_min, float v_max)
```

Et tracer la fonction  $t \mapsto \cos(t)$  sur  $[0, \pi]$  sous Python avec un pas de  $\pi/n$  où  $n = 1000$ .

En fait aucun des kernels précédents n'utilise vraiment la puissance de calcul du GPU : le nb de calcul est trop faible et la vitesse d'exécution est limitée par la lecture et l'écriture des données dans les tableaux.

**Exo 4** Pour  $s \in ]1, \infty]$ , on note  $\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$  la fonction zeta et pour  $M \geq 1$ , son approximation  $\zeta_M(s) = \sum_{n=1}^M \frac{1}{n^s}$ .

On veut tracer l'approximation  $s \mapsto \zeta_M(s)$  pour  $s \in [2, 4]$ ,  $M$  quelconque, et en prenant  $n$  points en abscisse.

Pour cela créer un kernel qui rempli un tableau de taille  $n$  avec les valeurs de  $\zeta_M(2 + 2 * i/n)$  pour  $i = 0, \dots, n - 1$  (bien sûr  $[2, 4]$  est arbitraire, si on veut on peut faire un kernel qui donne les valeurs dans un certain  $[v_{min}, v_{max}]$ ).

```
--global__ void tableau_zeta (int n, float * a, int M)
```

On a intérêt à créer une fonction sur le "device" qui calcule  $\zeta_M(x)$  :

```
--device__ float zeta (int M, float x)
```

Rq : la fonction CUDA pour calculer  $x^y$  est  $powf(x, y)$ , cf <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

section E

## Grille de blocs/ threads de dimension 2 ou 3

- On peut créer des grilles de blocs et de threads en dimension 1,2 ou 3. Cuda appelle les axes  $x$ ,  $y$ ,  $z$ .
- Bien sûr (théoriquement) les grilles en dim 1 suffiraient : un grille en dimension 2 se décrit en dim 1 en mettant les lignes bout à bout..
- Pratique par exemple si on manipules des matrices : la grilles des threads (en dim 2) a la même forme que la matrice.

## Exemple : appliquer une même fonction aux coefficients d'une matrice

- A matrice avec  $Nb\_ligne$  lignes  $Nb\_col$  colonnes. On veut calculer  $B$  avec  $B_{i,j} = f(A_{i,j})$  pour  $f$  une certaine fonction.
- Lancer (au moins)  $Nb\_ligne \times Nb\_col$  threads.
- Choix de la taille des blocs : on choisit des bloc de taille  $BlockDim.x \times BlockDim.y$ , par exemple  $16 \times 16$  (cela fait 256 threads donc sous la limite 1024)
- Choix de la grille. On identifie l'axe  $x$  aux colonnes et  $y$  aux lignes :

Grille de blocs de taille

$gridDim.x = \text{entier}(Nb\_col / blockDim.x) + 1$  selon l'axe x

Grille de blocs de taille

$gridDim.y = \text{entier}(Nb\_ligne / blockDim.y) + 1$  selon l'axe y

- Chaque threads connaît :

Le numero selon l'axe  $x$  de son bloc : `blockIdx.x`

Le numero selon l'axe  $y$  de son bloc : `blockIdx.y`

Sa position selon l'axe  $x$  dans son bloc : `threadIdx.x`

Sa position selon l'axe  $y$  dans son bloc : `threadIdx.y`

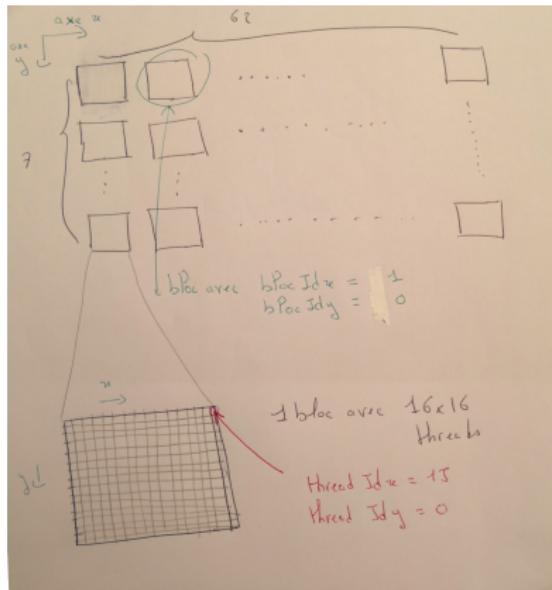
- Exemple A matrice 100 lignes et 1000 colonnes. Et bloc de taille  $16 \times 16$ .

Alors : Grille de bloc de taille 7 selon  $y$ , et 63 selon  $x$

$\text{blockIdx.x} \in \{0, \dots, 62\}$ ,  $\text{blockIdx.y} \in \{0, \dots, 6\}$

$\text{threadIdx.x} \in \{0, \dots, 15\}$ ,  $\text{threadIdx.y} \in \{0, \dots, 15\}$ ,

# La grille de threads en dessin



Position d'un thread :

$$ind\_x = blockIdx.x * blockDim.x + threadIdx.x$$

$$ind\_y = blockIdx.y * blockDim.y + threadIdx.y$$

## Le kernel qui applique une fonction à un tableau

```
--global__ void appli_fonc ( int nb_lig , int nb_col ,  
                           float * A, float * B ) {  
  
    int ind_col=threadIdx.x+blockDim.x*blockIdx.x;  
    int ind_ligne=threadIdx.y+blockDim.y*blockIdx.y;  
    int global_index;  
    if ((ind_col >= nb_col) || (ind_ligne >= nb_ligne) )  
        {return ;};  
    position=ind_ligne*nb_col+ind_col;  
    B[position]=function_a_appli(A[ position ]);  
}
```

Rq : on accède au tableau A de dimension 2 adressé par un pointeur comme à un tableaux de dim 1 dont les lignes sont mises bout à bout en mémoire (convention du C) ce qui donne :

$A[position]$  avec,  $position = ind\_ligne * nb\_col + ind\_col$

Le code Python se trouve dans le fichier `matrice_fonction.py`.  
Le kernel (qui se trouve dans le fichier `mat_fun.cu`) est lancé par

`BLOCKDIMx=16`

`BLOCKDIMy=16`

`GRIDDIMx=nbcoll //BLOCKDIMx+1`

`GRIDDIMy=nbligne //BLOCKDIMy+1`

```
applique_func(np.int32(nbligne), np.int32(nbcoll),  
              input_gpu, output_gpu,  
              block=(BLOCKDIMx, BLOCKDIMy, 1),  
              grid=(GRIDDIMx, GRIDDIMy, 1))
```

## Un exemple classique : ensemble de Mandelbrot

C'est l'ensemble  $\mathcal{M}$  des points  $c$  du plan complexe tels que la suite  $(z_n)_{n \geq 1} \in \mathbb{C}^{\mathbb{N}}$  définie par la récurrence suivante soit bornée :

$$z_{n+1} = z_n^2 + c, z_0 = 0.$$

On 'trace' l'ensemble de Mandelbrot, de la manière suivante :

On se donne un seuil  $S > 0$  et un nb d'itération maximal  $N_{max}$  (p.ex.  $N_{max} = 10^4$  ).

Pour chaque point  $c$  du plan complexe :

- on calcule les itérées de la suite  $(z_n)_n$  et on s'arrête dès que  $|z_n|^2 > S$  ou bien si l'indice  $n$  atteint  $N_{max}$ ,
- on associe à ce point  $c$ , le niveau de couleur  $n/N_{max} \in [0, 1]$ .

On sait que si la suite atteint le module 2 alors elle divergera, et donc un seuil  $S = 4$  est souvent choisi.

# Implementation Cuda

- Sur la portion du plan complexe

$K = \{z \in \mathbb{C} \mid |Re(z)| < 2, |Im(z)| < 2\}$  on trace l'ensemble de Mandebrot par l'algorithme précédent, en découplant l'ensemble  $K$  en une grille de point.

- Chaque thread s'occupe de calculer la couleur d'un point de la grille.
- Le fichier Python est `matrice_mandel.py` et le kernel est dans `mat_fun.cu`.

*Il y a aussi une animation qui montre la déformation de l'ensemble en faisant varier les valeurs du seuil (si l'animation ne lance pas sous Spyder, la lancer sous ipython3).*

- Exercice : comprendre le code et lancer.

## Exercice : ensemble de Julia

On se donne  $c \in \mathbb{C}$ . L'ensemble de Julia est l'ensemble  $\mathcal{J}$  des points  $z$  du plan complexe tel que la suite  $(z_n)_{n \geq 1} \in \mathbb{C}^{\mathbb{N}}$  définie par la récurrence suivante soit bornée :

$$z_{n+1} = z_n^2 + c, z_0 = z.$$

Rq : pour chaque  $c$  différent, l'ensemble de Julia est différent.

### Exercice :

Tracer l'ensemble de Julia pour  $c = 0.3 + 0.5i$  (*Utiliser le kernel et code Python servant à tracer l'ensemble de Mandelbrot et modifier..*)

# Méthode de Monté Carlo sous CUDA

- Les méthodes de Monté Carlo sont intensives en calculs.
- Le nb d'itérations doit être grand en général.
- Elles s'implémentent efficacement sous GPU.

# Générateurs de nombres aléatoires

- Des générateurs de 'nombres aléatoires' rapides n'existent pas.
- Les ordinateurs génèrent des suites de nombres 'pseudo-aléatoires' :  
il s'agit de suites déterministes qui ont l'apparence et certaines propriétés des suites i.i.d. Par exemple, elles vérifient la Loi des Grands Nombres...
- Les algorithmes de création de suites pseudo-aléatoires sont multiples. Un type courant est l'utilisation de suites récurrentes bien choisies (ex : fonction `drand48()` en C).
- Une suite pseudo-aléatoire donne toujours le même suite de valeurs (vu qu'elle est déterministe!). Mais souvent la suite dépend d'un paramètre ('seed'=la graine) que l'on peut changer : si on change la valeur de la seed alors on change les valeurs de la suite. Par ex. pour des suites pseudo-aléatoire définies par récurrence, la seed est le point de départ de la suite.

# Générateurs de nombres aléatoires sous CUDA

- Pour paralléliser le code, il faut avoir des générateurs de nombres aléatoires qui peuvent fonctionner en parallèles.

- Les algorithmes pour cela sont complexes. Une description de certains algorithmes peut être trouvée sur

[https://www.thesalmons.org/john/random123/releases/  
latest/docs/index.html](https://www.thesalmons.org/john/random123/releases/latest/docs/index.html)

- CUDA implémente certains de ces algorithmes au travers de la library cuRAND

<https://docs.nvidia.com/cuda/curand/index.html>

# cuRAND

Les générateurs de nb aléatoires de la librairie cuRAND fonctionnent de la manière suivante :

- Etape 1 : On initialise le générateur. Pour cela il faut spécifier trois variables :
  - Le seed : on peut prendre seed = 0.
  - Le numéro de la suite appelé subsequence : pour chaque numéro différent, le générateur tire une suite i.i.d. indépendante des suites avec d'autres numéros.  
Notons  $(Z_n^i)_{n \geq 0}$  la suite pseudo-aléatoire numéro  $i$ .  
On utilisera des suites avec des numéros différents pour chaque thread, si on veut que chaque thread utilise des suites i.i.d. indépendantes.
  - Le decalage offset : Si on veut que la suite  $(Z_n^i)_{n \geq 0}$  commence à  $n = \text{offset}$  plutôt que depuis  $n = 0$
- Etape 2 : On appelle le générateur pour obtenir une réalisation d'une v.a. distribuée uniformément sur  $[0, 1]$  (standard Gaussien est possible aussi)

Rk 1 : En fait, à  $i$  fixé, les suites  $(Z_n^i)_n$  sont de longueurs finies égales à  $2^{64}$  :  $(Z_n^i)_{0 \leq n \leq 2^{64}-1}$ .

Rk 2 : Il y a  $2^{64}$  numéros de suites différentes

Rk 3 : Cela fait  $2^{64}$  suites aléatoires de longueur  $2^{64}$  (soit  $2^{128}$  nombres !)

## Exemple : calcul $\pi$ par Monté-Carlo

Soit  $(U_i)_{i \geq 1}$  une suite i.i.d de variable aléatoire à valeur dans  $\mathbb{R}^2$  et de loi uniforme sur le carré  $[-1, 1] \times [-1, 1]$ .

Notons  $D$  le disque de rayon 1. On a  $P(U_1 \in D) = \frac{\pi}{4}$ .

Alors par LGN :

$$\frac{1}{n} \sum_{i=1}^n 1_D(U_i) \xrightarrow{n \rightarrow \infty} \frac{\pi}{4}.$$

Cela permet d'estimer  $\pi$  par méthode de Monté-Carlo.

Il faut un nb d'itérations  $n$  le plus grand possible...

### Méthode de parallélisation :

On choisit  $n = M \times N$ , ( $M$  et  $N$  grands).

- On lance  $M$  threads
- Chaque thread calcule indépendamment par Monté Carlo une approximation de  $\pi$  en utilisant  $N$  variables aléatoires.
- On fait la moyenne des résultats des  $M$  threads.

# Examinons le kernel

```
-global__ void counthits(int n, uint *hitsp,  
                         unsigned decalage_index) {  
  
    curandStatePhilox4_32_10_t state;
```

Créer une variable 'state' qui va contenir l'état du générateur aléatoire

```
int index = threadIdx.x + blockIdx.x * blockDim.x;  
unsigned hits = 0;  
float x1,y1,x2,y2;  
curand_init(0, index+decalage_index, 0, &state);
```

Initialise le générateur aléatoire avec :

seed =0

Numéro de la suite = numéro du thread (+ un décalage mis en paramètre si on veut relancer en utilisant d'autres numéros de suites)

Décalage de la suite =0

```
float4 rand_vec;
```

Créer une variable de type vecteur de 4 float qui va servir à contenir 4 tirages de v.a.

```
int tries=0;  
while (tries < n)  
{  
    rand_vec=curand_uniform4 (&state );
```

La ligne qui dit appelle curand pour générer 4 tirages de v.a. uniforme sur [0,1]

```
x1 = 2*rand_vec.x-1;  
y1 = 2*rand_vec.y-1;  
x2 = 2*rand_vec.z-1;  
y2 = 2*rand_vec.w-1;  
if ( (x1*x1 + y1*y1) < 1) { hits++;}  
if ( (x2*x2 + y2*y2) < 1) { hits++;}  
tries=tries+2;  
}  
hitsp[index]=hits;
```

Sauve le nb de fois où la variable est tombée dans le cercle de rayon 1 dans le tableau des résultats

```
}
```

- Les codes Python et le kernel sont `pi_cuda.py` et `pi_kernel.cu`.

On peut apprécier la rapidité d'exécution...

# Exercice : générateurs de v.a. de loi Gamma

## But

- L'objectif est de créer un kernel qui génère un grand nombre de variables de loi Gamma et les stocke dans un tableau.
- La méthode pour générer une v.a. de loi Gamma est la méthode de rejet.
- Chaque thread générera une seule variable aléatoire de loi Gamma par méthode de rejet et la stockera dans le tableau.
- Voir feuille distribuée pour les détails mathématiques sur la méthode de rejet.