

TP 3-4

Master 2 SID
Benoist GASTON
benoist.gaston@univ-rouen.fr

CUDA prise en main

L'ensemble des TPs sont proposés sous forme de notebook (<https://github.com/benoistgaston/m2sid-2021.git>).

Les notebooks peuvent être joués sur l'environnement colab research de google :

<https://colab.research.google.com>

Nécessite un compte google.

Addition de matrice : `tp01-addmat.ipynb`


L'objectif est d'additionner deux matrices terme à terme sur GPU. On considérera des threads organisés en 1 seul bloc 2D.

1. Compléter le kernel `add_mat` afin que chaque thread prenne en charge une et une seule addition
2. Compléter les transferts mémoire entre le host et le device
3. Appeler le kernel pour les matrices `g_A` et `g_B` en utilisant un bloc correctement dimensionné

On se propose de traiter des matrices plus grosses

4. Sur la base du kernel `add_mat` créer un kernel `add_huge_mat` dans lequel chaque thread prend en charge `N` additions terme à terme
5. Appeler le kernel sur les matrices `g_HA` et `g_HB` en utilisant un bloc correctement dimensionné

Addition de matrice

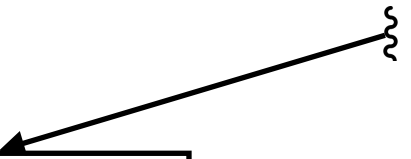
$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \dots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \dots & a_{mn} + b_{mn} \end{bmatrix}$$


Multiplication de matrice : `tp02-multmat.ipynb`

L'objectif est de réaliser une multiplication de deux matrices terme à terme sur GPU. On considérera des threads organisés en bloc 1D ou 2D et en grille 1D ou 2D.

1. Compléter le kernel `mult_mat` afin que chaque thread prenne en charge une boucle interne.
2. Appeler le kernel pour les matrices `g_a` et `g_b` en utilisant des blocs et des grilles correctement dimensionnés.

```
for i in range(dim1):  
    for j in range(dim2):  
        for k in range(dim3):  
            c[i][j] += a[i][k] * b[k][j]
```



CUDA prise en main

Addition de matrice

Produit scalaire N-vecteur : tp03-dotvec.ipynb

Objectif : réaliser le produit scalaire de deux vecteurs 1D sur GPU. Le produit scalaire est donné par la formule et l'algorithme ci contre.

Dans un premier temps on ne réalisera sur GPU que les produits terme à terme de chaque vecteur ; **la somme est réalisée sur le host**.

1. Écrire le kernel `dot_vec` afin que chaque thread prenne en charge une et une seule multiplication terme à terme.
2. Appeler le kernel pour les vecteurs `g_va` et `g_vb` en utilisant un seul bloc correctement dimensionné.

Réfléchir à un algorithme permettant de réaliser la somme finale sur le device:

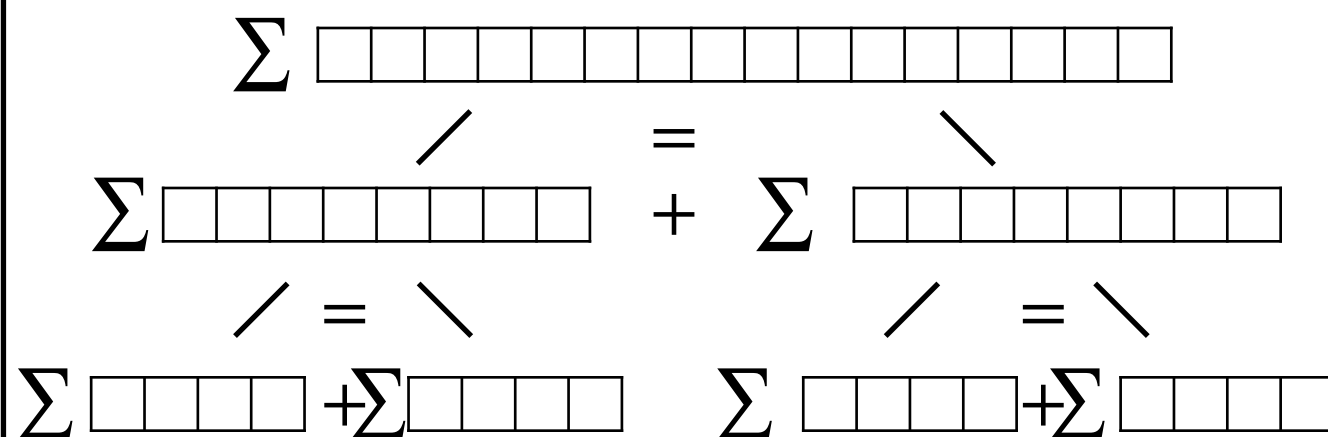
- Une première solution consiste à réaliser la somme par un seul thread.
- Une seconde solution peut s'inspirer du schéma ci-contre.

Note: on peut opérer si une synchronisation (barrière) des threads d'un bloc par la fonction `cuda : __syncthreads()`

3. Sur la base du kernel `dot_vec` créer un kernel `complete_dot_vec` qui permet de réaliser les produit terme à terme et la somme finale.
4. Tester les différents algorithmes.
5. Selon vous, quelle sont les limites de ces méthodes de réduction sur GPU ?

$$[a_1, a_2, \dots, a_n] \times [b_1, b_2, \dots, b_n] = a_1b_1 + a_2b_2 + \dots + a_nb_n$$

```
res=np.empty_like(va)
for i in range(va.size):
    res[i]=va[i]*vb[i]
dotprod=sum(res[:])
```



CUDA Game Of Life

Jeu de la vie : tp04-conway.ipynb

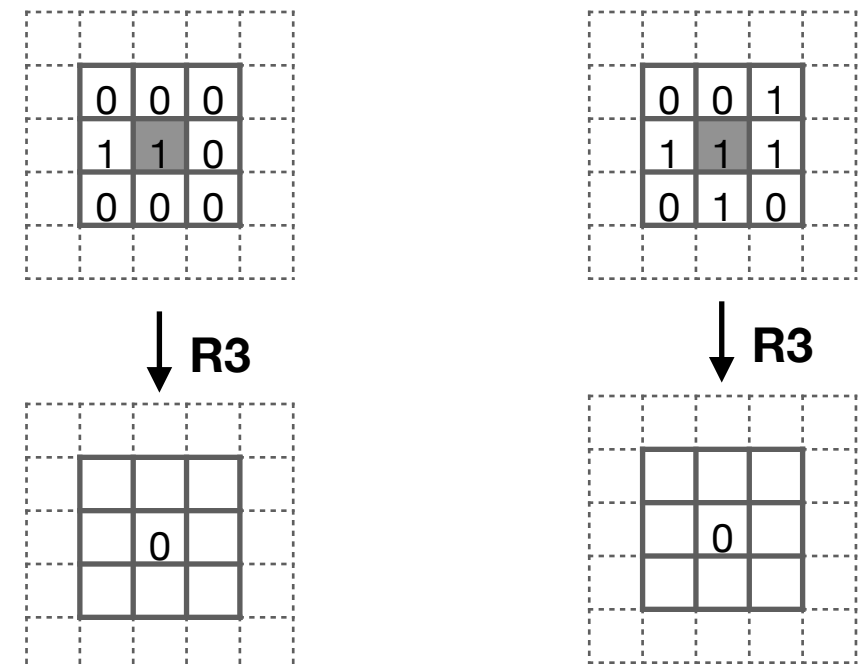
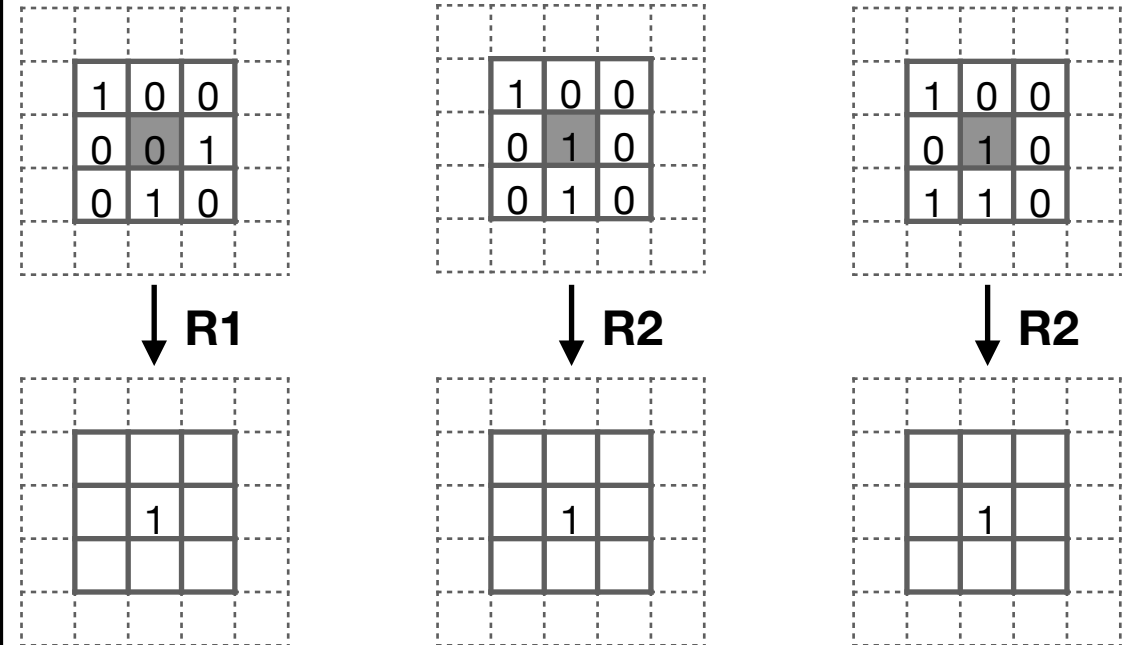
Objectif : réaliser le jeu de la vie sur GPU.

Rappel

- Les jeux de la vie, ou automates cellulaires, sont définis sur une grille de cellules. Les cellules sont dans un état donné (mort ou vivant). L'état des cellules évolue dans le temps en fonction de l'état des cellules voisines selon des règles simples.

Règles de base

- Etat 0 ou 1, i.e. morte ou vivante
- R1 : une cellule morte possédant exactement 3 voisins (vivants) naît, i.e. Etat 0->1
- R2 : une cellule vivante possédant 2 ou 3 voisins (vivants) reste vivante, i.e. Etat 1->1
- R3 : une cellule vivante qui possède moins de 2 voisins (vivants) ou plus de 3 voisins (vivants) meurt par isolement ou surpeuplement i.e. Etat 1->0



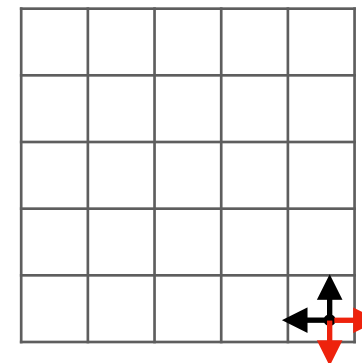
CUDA Game Of Life

TP

- Le notebook contient les fonctions suivantes :
 - init_grid(n)** : qui initialise une grille aléatoire de dimension nXn (pour des raisons de reproductibilité la graine aléatoire est fixée à 0 ;
 - enlarge_grid(grid)** : qui élargie une grille pour simplifier la gestion des bords ;
 - evolution(grid)** : qui prend en entrée une grille élargie grid et retourne le résultat de l'application des règles du jeu de la vie sur la partie interne de la grille ;
 - gamelife(grid,n)** qui a partir d'une grille joue n itération du jeu de la vie.

Questions

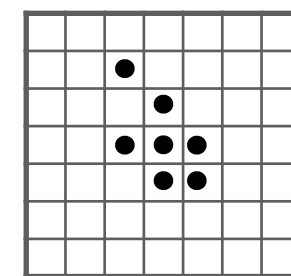
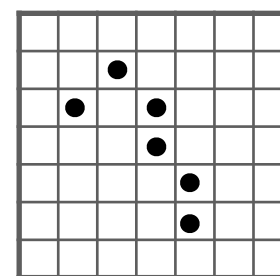
- Compléter le kernel `conway(int* g_output,int* g_input)` afin que chaque thread calcul l'évolution d'une cellule de la grille initiale.
- Compléter la fonction `g_gamelife(grid,n)` afin d'appeler le kernel `conway` sur des grilles de jeu de 32x32 (un bloc 32x32).
- Désactiver l'affichage (utilisation de matplotlib) et comparer les temps d'exécution avec `%timeit`.
- Le `g_output` d'une itération correspond au `g_input` de l'itération suivante. Sur ce constat, proposer une amélioration afin de réduire les transferts mémoire entre host et device.
- Proposer une évolution pour traiter des grilles de jeu plus grandes.



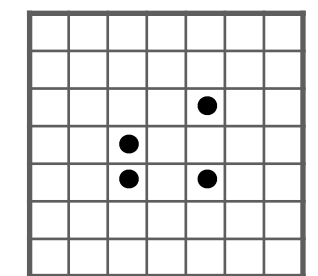
0	0	0	0	0	0	0
0						0
0						0
0						0
0						0
0						0
0	0	0	0	0	0	0

Élargissement : gestion des bords

ité 1 : input ité 1 : output



ité 2 : input



ité 2 : output