# Programmation GPU

(2023-2024)

# Projet

## Wave Function Collapse sur GPUs

julien.jaeger@cea.fr
mael.martin@cea.fr

Les objectifs de ce projet sont :

- ❒ écrire une application pour GPU en utilisant CUDA ou OpenMP avec les constructions target
- ❒ mettre en œuvre les optimisations GPU vues en cours
- ❒ réaliser une étude de performance en comparant l'application écrite sur GPU avec une implémentation OpenMP parallèle sur CPU

## I Déscription

**Carrés latins -** Les carrés latins sont des tableaux carrés de côtés $n$. Les $n$ lignes et $n$ colonnes sont remplies de $n$ éléments distincts. Un exemple de carré latin et celui de Sator qui contient des palindromes d'une phrase de 5 mots de 5 lettres chacuns. Il est aussi possible de remarquer qu'en permutant deux lignes ou deux colonnes d'un carré latin, on obtient encore un carré latin : il n'y a pas unicité des carré latin pour une dimension $n$ donnée.

| | | | | |
|---|---|---|---|---|
| S | A | T | O | R |
| A | R | E | P | O |
| T | E | N | E | T |
| O | P | E | R | A |
| R | O | T | A | S |

Table 1 – Carré de Sator

**Sudoku -** Le sudoku est une forme de carré latin. Pour une taille $k$, un sudoku est un tableau carré de côté $3k$, composé de $k^2$ régions carrées de côtés $k$ chacune. Chacune de ces régions doit contenir $k^2$ éléments distincts. Chaqune des $k^2$ ligne et $k^2$ colone est composé de $k^2$ éléments disctincts. En plus des lignes et colones, les deux diagonales du sudoku doivent contenir $k^2$ éléments distincts.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 5 | 6 | 3 | 8 | 9 | 7 | 2 |
| 3 | 6 | 2 | 4 | 7 | 9 | 1 | 8 | 5 |
| 7 | 8 | 9 | 2 | 1 | 5 | 3 | 6 | 4 |
| 9 | 2 | 6 | 3 | 4 | 1 | 7 | 5 | 8 |
| 1 | 3 | 8 | 7 | 5 | 6 | 4 | 2 | 9 |
| 5 | 7 | 4 | 9 | 8 | 2 | 6 | 3 | 1 |
| 2 | 5 | 7 | 1 | 6 | 4 | 8 | 9 | 3 |
| 8 | 4 | 3 | 5 | 9 | 7 | 2 | 1 | 6 |
| 6 | 9 | 1 | 8 | 2 | 3 | 5 | 4 | 7 |

Table 2 – Exemple de sudoku

**Wave Function Collapse -** En physique quantique, une fonction d'onde (*wave function* en anglais) est une description mathématique d'un système quantique isolé. Une fonction d'onde peut décrire la probabilité qu'une particule soit à une position donnée. Cette fonction d'onde peut s'effondrer (*collapse* en anglais) due à une intéraction avec l'extérieur du système. Il est possible de reprendre cette idée pour générer de manière procédurale des ressources ou résoudre certains problèmes sous contraintes.

**Domaine de résolution -** Nous allons appliquer l'algorithme Wave Function Collapse (WFC) dans une grille carré pour ce projet. À chaque case du domaine considéré sont associés des états possibles (des nombres dans le cas du sudoku). Chaque case a une entropie associée, plus celle-ci est grande, moins nous pouvons savoir l'état de la case avec certitude. Dans ce problème une case d'entropie nulle est une case dont l'état est connue. Le principe de l'algorithme est de trouver la case avec la plus petite entropie, de faire s'effondrer la case, de mettre à jour les possibilités des cases voisines.

```
type etats      = set<int>
type domaine<N> = vector<N,N,etats>
var  D: domaine
var  seed: int

func entropy(etats) -> int
func find_min_entropy() -> Maybe<(int,int)>
func set_random_seed(int) -> ()
func collapse_state(etats) -> int
func update_domaine(domaine,int,int) -> bool

begin WFC:
    set_random_seed(seed)
    while Some(x,y) = find_min_entropy(D):
        assert(entropy(D[x,y]) != 0)
        D[x,y] = collapse_state(D[x,y])
        if not update_domaine(D,x,y):
            return "propagation failed"
    return D
```

Listing 1 – Algorithme WFC

Dans le listing ci-dessus est décrit l'algorithme en question sous forme de pseudo-code. Le domaine `D` est initialisé en fonction du jeu d'entrée. Tant qu'il existe une entropie minimale non nulle nous allons faire s'effondrer l'état d'une case puis mettre à jour le reste du domaine. Lors de la mise à jour du domaine il se peut qu'une case n'ait plus d'état possible, dans ce cas l'algorithme a échoué.

Il est important de noter que plusieurs cases peuvent avoir la même entropie, dans ce cas il faut choisir la case «au hazard». De même, lors de l'effondrement d'une case, l'était choisi doit l'être «au hazard». Les nombres choisis «au hazard» le sont en utilisant un générateur de nombre pseudo-aléatoire initialisé par une «seed» avant chaque début de résolution. Pour que les résultats de l'algorithme WFC soient consistant quel que soit le solveur choisi, il est important de maitriser ce générateur aléatoire.

**Organisation du code source -** Le code source de ce projet est disponible dans l'archive du projet. Cette archive contient :

- ❒ `src`, `inc` : les sources du projet, c'est ce code que vous devrez modifier et adapter.
- ❒ `data` : des jeux de donnée pour configurer les grilles à résoudre, quel que soit le solveur choisi les résultats doivent être identiques. Ce dossier peut être enrichi par vos propres exemples.
- ❒ `CMakeLists.txt` : le fichier de configuration pour cmake. Il n'a pas besoin d'être modifié.

Pour configurer la compilation du projet, utiliser `cmake` (ne pas utiliser le dossier des sources en tant que dossier de compilation...) Pour compiler le projet vous pouvez utiliser `make` ou `ninja` en fonction des options passés à `cmake`. Une aide est fournie avec le flag `-h` : `./build/wfc -h`.

**Format des fichiers en entrée -** Il existe deux type de fichiers d'entrée pour la partie obligatoire du projet. La première ligne indique s'il s'agit de la déscription d'une région simple ou d'une grille de sudoku ainsi que la taille des grilles. Sur chaque ligne suivante est indiquée l'état d'une case de la grille sous la forme `coordonnées=état`. Par exemple :

```
$ cat data/simple-5x5.data
s5
0,0,0,1=4
0,0,2,3=1

$ cat data/grid-3x3.data
g3
0,0,0,1=4
0,0,2,2=1
1,1,0,1=4
2,2,2,2=1
```

**Format des fichiers en sortie -** Respectez bien le format des fichiers pour la sauvegarde des résultats, ils serviront à valider la correction de votre implémentation. Les fichiers sont composés d'une entête avec les dimensions de la grille (racine carrée du nombre de région du sudoku) puis des blocks de cette grille (côté d'une région du sudoku). Ensuite sont les valeurs des cases stockées linéairement, une par ligne par région. Par exemple :

```
$ cat output/grid-2x2.123.save
grid:  2
block: 2
1 2 3 4
3 4 1 2
2 1 4 3
4 3 2 1

$ cat output/simple-2x2.456.save
grid:  1
block: 2
1 2 3 4
```

# II Travail à réaliser

**Groupes -** Le projet s'effectue en **binome**.

**Attentes -** Nous attendons au moins deux implémentations de l'algorithme WFC pour résoudre des sudoku d'une taille arbitraire définie lors de la compilation du programme. Une limite intéressante peut être 8, vous devrez expliquer pourquoi et proposer des solutions pour passer outre cette limitation. Notez que le résultat devra rester constitant avec le flag `-p1` quel que soit le solveur choisi. Il peut être intéressant de pouvoir partager différentes tentatives de résolution entre le CPU et le GPU.

1. La première implémentation sera écrite en OpenMP pour être exécuté sur CPU.
2. La seconde sera écrite en CUDA ou OpenMP avec l'aide des targets pour exécuter le code sur GPU.

Une progression conseillée pour l'implémentation est la suivante : résoudre des régions simples de sudoku (fichiers `simple-*.data`) ; résoudre des sudoku sans la contrainte des deux diagonales (fichiers `grid-*.data`) puis rajouter la contrainte des diagonales.

**Bonus -** L'implémentation de génération de tilemaps est optionnelle, si elle est réalisée elle ne devra l'être uniquement sur GPU avec CUDA ou OpenMP target au choix. Voir la section suivante.

**Soutenance -** La soutenance aura lieux le **31 janvier**. Une archive **tar** de votre code source devra être envoyée par mail avant le **28 janvier** minuit. Lors de la soutenance nous souhaiterons que les points suivants soient abordés :

- ❒ description, motivation des stratégies de parallélisation employées ainsi que les opportunités de parallélisation présentes dans le problème.
- ❒ choix d'implémentations et difficultées rencontrées pour les différents modèles de programmation OpenMP CPU, OpenMP target, CUDA.
- ❒ étude de performance des implémentations et analyse des résultats obtenus.
- ❒ pistes d'amélioration possible des implémentations proposées.

La soutenance sera composée de **10 minutes de présentation** suivies de **10 minutes de question**. Nous évaluerons en premier lieux la démarche scientifique, ce qui signifie que nous attachons une plus grande importance à la présentation de la démarche et des résultats qu'aux résultats en eux-même.