Compte Rendu : Implémentation du Game of Life en CUDA

Hugo

8 Février 2026

1 Introduction

Ce document présente l'implémentation du noyau CUDA pour le Jeu de la Vie et les optimisations testées.

2 Validation et Configuration

L'exactitude de l'implémentation a été vérifiée en comparant les résultats du GPU avec une implémentation de référence en PyTorch. L'exécution de la commande suivante confirme la correspondance des résultats :

```
python conway.py test
> Both implementations match
```

La configuration matérielle utilisée pour les tests est la suivante :

```
NVIDIA-SMI 535.288.01 Driver Version: 535.288.01 CUDA Version: 12.2
GPU: NVIDIA GeForce GTX 1070 (8192MiB)
```

3 Performances et Optimisations

Une optimisation préliminaire a consisté à utiliser le type `unsigned char` (1 octet) au lieu de `int` (4 octets) pour stocker les états des cellules. Cette modification permet de diviser par quatre l'empreinte mémoire de la grille et d'optimiser l'utilisation de la bande passante mémoire ainsi que du cache.

L'implémentation actuelle utilise également des **grid-stride loops** pour permettre au noyau de traiter des grilles de n'importe quelle taille, indépendamment de la taille de la grille de blocs CUDA. Cela améliore la robustesse et permet une meilleure réutilisation des threads.

Plusieurs versions du noyau ont été testées pour l'application des règles du Jeu de la Vie.

3.1 Version 1 : Expression Booléenne Directe

Cette version utilise une expression logique concise pour déterminer l'état suivant :

```
new_grid[idx] = alive_neighbors == 3 || (grid[idx] == 1 &&
    alive_neighbors == 2);
```

Performance obtenue : **1001 FPS**.

3.2 Version 2 : Branchements Conditionnels

Cette version utilise des structures `if/else` plus explicites (bien que logiquement redondantes dans ce cas précis) :

```
new_grid[idx] = 0;
if (grid[idx] == 0 && alive_neighbors == 3 ||
     (grid[idx] == 1 && (alive_neighbors == 4 || alive_neighbors == 3)))
    {
     new_grid[idx] = 1;
}
else if (grid[idx] == 1 && (alive_neighbors == 4 || alive_neighbors ==
    3)) {
     new_grid[idx] = 1;
}
```

Performance obtenue : **1057.26 FPS**.

3.3 Version 3 : Mémoire Partagée (Shared Memory)

L'optimisation repose sur l'utilisation de la *Shared Memory*. L'idée est de réduire les accès à la mémoire globale lente en chargeant une tuile de données dans une mémoire locale rapide partagée par les threads d'un même bloc.

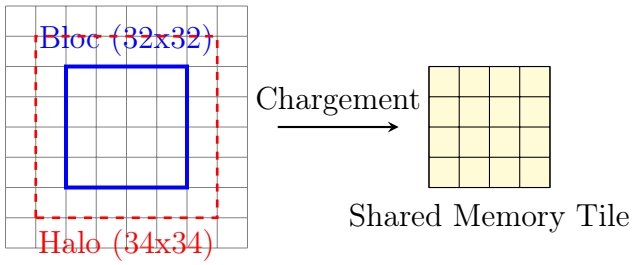

Figure 1: Schéma du chargement de la tuile et du halo en mémoire partagée.

Chaque bloc de 32x32 threads charge une zone de 34x34 (*halo* de 1 pixel) pour permettre le calcul des voisins sur les bords sans accès mémoire globale supplémentaire.

L'implémentation du noyau utilisant la mémoire partagée est la suivante :

```
__global__ void game_of_life_kernel(unsigned char *grid, unsigned char *
    new_grid, int width, int height) {
     __shared__ unsigned char tile[34][34];
     int tx = threadIdx.x; int ty = threadIdx.y;
     int x = blockIdx.x * 32 + tx; int y = blockIdx.y * 32 + ty;

     // Chargement collaboratif avec halo
     int tid = ty * 32 + tx;
     int block_start_x = blockIdx.x * 32 - 1;
```

```
    int block_start_y = blockIdx.y * 32 - 1;

    for (int i = tid; i < 34 * 34; i += 1024) {
        int ly = i / 34; int lx = i % 34;
        int gx = block_start_x + lx; int gy = block_start_y + ly;
        unsigned char val = 0;
        if (gx >= 0 && gx < width && gy >= 0 && gy < height)
            val = grid[gy * width + gx];
        tile[ly][lx] = val;
    }
    __syncthreads();

    if (x < width && y < height) {
        int sx = tx + 1; int sy = ty + 1;
        int neighbors = tile[sy-1][sx-1] + tile[sy-1][sx] + tile[sy-1][
    sx+1] +
                        tile[sy][sx-1]   +                  tile[sy][sx
    +1] +
                        tile[sy+1][sx-1] + tile[sy+1][sx] + tile[sy+1][
    sx+1];
        unsigned char current = tile[sy][sx];
        new_grid[y * width + x] = (neighbors == 3) | (current & (
    neighbors == 2));
     }
}
```

Performance obtenue : **1165.18 FPS**.

3.4 Version 4 : Lookup Table (LUT) et Vectorisation

Pour cette version, nous changeons de paradigme. Au lieu de calculer l'état futur cellule par cellule, nous utilisons une table de correspondance (Lookup Table) précalculée.

L'idée est qu'un bloc de 2×2 cellules (4 pixels) dépend entièrement de l'état d'un bloc de 4×4 cellules qui l'entoure. Un bloc de 4×4 contient 16 cellules, ce qui correspond exactement à un entier de 16 bits ($2^{16} = 65536$ possibilités).

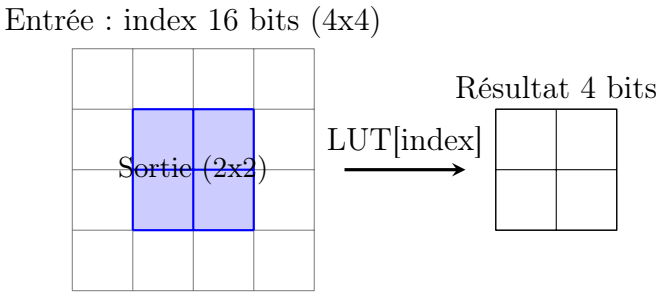

Figure 2: Principe de la LUT : Un voisinage 4x4 détermine le futur des 4 cellules centrales.

Nous précalculons donc une table de 65536 entrées en mémoire globale. Chaque entrée contient les 4 bits de résultat pour le bloc central.

Le noyau CUDA est modifié pour que **chaque thread traite désormais 4 pixels** (un bloc 2×2). Le nombre de threads est donc divisé par 4, réduisant le surcoût de lancement et augmentant l'intensité arithmétique (ici remplacée par des opérations sur les bits).

```
// Extrait du kernel
```

```
int tile_r = 2 * ty;
int tile_c = 2 * tx;
uint16_t state_idx = 0;

// Construction de l'index 16 bits depuis la shared memory
#pragma unroll
for (int r = 0; r < 4; r++) {
    for (int c = 0; c < 4; c++) {
        if (tile[tile_r + r][tile_c + c]) {
            state_idx |= (1 << (r * 4 + c));
        }
    }
}

// Lecture unique en memoire pour 4 pixels
unsigned char res = lut[state_idx];

// Ecriture des 4 pixels
new_grid[...] = (res >> 0) & 1;
new_grid[...] = (res >> 1) & 1;
// ... (idem pour les 2 autres)
```

Performance obtenue : **2455.34 FPS**. C'est un gain considérable (+**110%** par rapport à la version shared memory) qui s'explique par la vectorisation du travail (1 thread = 4 pixels) et la suppression complète des branchements et des calculs arithmétiques au profit d'opérations bit-à-bit très rapides.

4 Analyse

De manière surprenante, la version avec branchements (Version 2) affiche une performance légèrement supérieure à la version booléenne compacte. Cependant, l'introduction de la **Shared Memory** (Version 3) a apporté un premier gain important. Enfin, l'approche **Lookup Table** (Version 4) écrase les précédentes en transformant le problème de calcul en un problème d'accès mémoire optimisé et vectorisé.

4.1 Version 5 : Maximisation de l'Occupation (1024 Threads)

L'optimisation précédente utilisait des blocs de 16×16 threads (256 threads) couvrant une zone de 32×32 pixels. Cependant, les GPU modernes supportent jusqu'à 1024 threads par bloc.

Nous avons donc augmenté la taille des blocs à 32×32 threads (1024 threads). Comme chaque thread traite toujours un carré de 2×2 pixels, chaque bloc CUDA couvre désormais une zone de 64×64 pixels sur la grille.

Avantages :

- Meilleure occupation du multiprocesseur (SM).
- Réduction du ratio *Halo / Données utiles*. Pour une tuile de 32^2 pixels, le halo est de $34^2 - 32^2 = 132$ pixels. Pour 64^2 pixels, le halo est de $66^2 - 64^2 = 260$ pixels. Le ratio halo/pixels passe de $132/1024 \approx 12.8\%$ à $260/4096 \approx 6.3\%$.

subsectionVersion 5 : LUT + uint16_t Dans cette version, on exploite les registres 64 bits du GPU. Chaque thread est responsable d'un entier `uint16_t`, qui représente un bloc de 4×4 cellules.

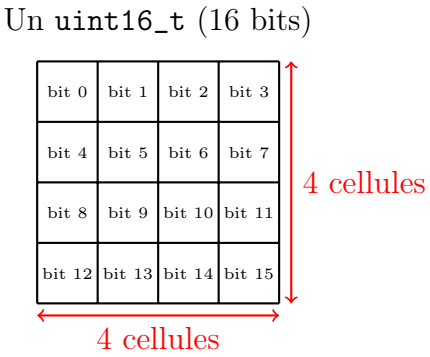

Figure 3: Représentation compacte : 16 cellules stockées dans un seul registre.

4.2 Le Problème du Voisinage

Pour calculer l'état futur de notre bloc 4×4 (Centre), nous avons besoin de connaître l'état de ses voisins immédiats (cellules bordures). Ce voisinage forme une grille de 6×6 cellules.

Les données nécessaires proviennent de **9 blocs** `uint16_t` différents : le bloc central et ses 8 voisins (Nord, Sud, Est, Ouest, et les diagonales).

4.3 L'Algorithme de "Super-Patch" 64 bits

Au lieu d'aller chercher les valeurs en mémoire à chaque calcul, nous construisons un objet virtuel temporaire dans les registres du thread : le **Patch**.

Un carré de 6×6 cellules correspond à 36 bits. Comme $36 < 64$, nous pouvons stocker l'intégralité du voisinage nécessaire dans un seul registre `uint64_t`.

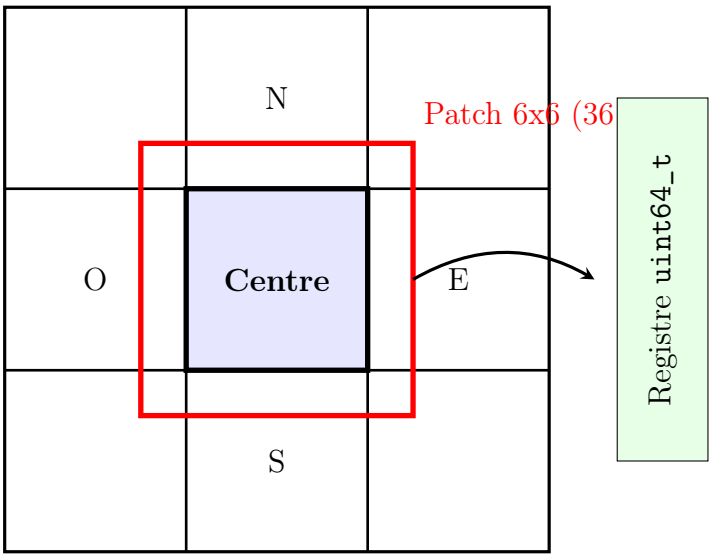

Figure 4: Construction du Patch : on extrait les bits nécessaires des 9 voisins pour former un seul mot de 64 bits.

Une fois ce `uint64_t patch` construit via des opérations bit-à-bit (masques et décalages), le thread n'a plus besoin d'accéder à la mémoire. Il contient toute l'information localement.

uint16 central (4x4)

15	12	13	14	15	12
3	0	1	2	3	0
7	4	5	6	7	4
11	8	9	10	11	8
15	12	13	14	15	12
3	0	1	2	3	0

Patch 6x6 extrait (avec couronne)

Figure 5: Numérotation des bits dans le bloc central 4x4 et la couronne 6x6 extraite par `build_patch6x6`.

Explication détaillée du code Le code effectue pour chaque ligne i de 0 à 5 :

- i == 0 : on construit la ligne au-dessus (row r-1) à partir des dernières lignes des blocs NO, N, NE. `mid4 = (N » 12) & 0xF` lit la dernière ligne du bloc N; `left_bit = (NO » 15) & 1` lit le bit le plus à droite (col 3) de la dernière ligne de NO; `right_bit = (NE » 12) & 1` lit le bit le plus à gauche (col 0) de la dernière ligne de NE.
- i == 1..4 : pour chaque ligne du centre on lit la ligne correspondante de `C` via `mid4 = (C » (4*ri)) & 0xF` (ri = i-1), et on récupère la colonne de gauche depuis `O` et la colonne de droite depuis `E` (bits isolés par des shifts appropriés).
- i == 5 : on construit la ligne sous le centre (row r+4) à partir des premières lignes de SO, S, SE (ex. `mid4 = (S » 0) & 0xF`).

Ces 6 lignes de 6 bits sont concaténées (`patch |= (uint64_t)row6 « (6 * i)`) pour former le mot 36 bits logé dans un `uint64_t`. Ensuite on extrait des fenêtres 4x4 par décalages successifs pour indexer la LUT.

4.4 Extraction et Lookup par Quadrants

Nous divisons ensuite notre problème 4×4 en quatre sous-problèmes 2×2 (quadrants). Pour chaque quadrant, nous extrayons une fenêtre de 4×4 bits depuis le patch 64 bits, ce qui nous donne un index pour interroger la LUT.

```
// Exemple conceptuel
uint64_t patch = build_patch6x6(N, S, E, O, ...);

// On extrait 4 fenetres depuis le patch
uint16_t idx_TL = extract(patch, 0, 0); // Top-Left
uint16_t idx_TR = extract(patch, 0, 2); // Top-Right
uint16_t idx_BL = extract(patch, 2, 0); // Bot-Left
uint16_t idx_BR = extract(patch, 2, 2); // Bot-Right

```

```
// Lookup instantanee
uint8_t res_TL = lut[idx_TL];
uint8_t res_TR = lut[idx_TR];
...

// Assemblage final
uint16_t result = (res_TL) | (res_TR << 2) | ...;
```

4.5 Avantages de cette méthode

1. **Réduction drastique des accès mémoire :** Une fois les blocs chargés en *Shared Memory*, tout le calcul se fait dans les registres.

2. **Zéro divergence :** Tous les threads exécutent exactement les mêmes opérations binaires, il n'y a aucun `if/else`.

3. **Parallélisme de bits :** En manipulant des `uint16` et `uint64`, nous traitons 16 cellules simultanément par thread avec un nombre d'instructions très faible.

Performance obtenue : 3200.59 FPS, soit un gain de plus de 30% par rapport à la version LUT précédente.

Résumé des performances :

Résultat mesuré La version actuelle du code (Lookup Table avec 1024 threads par bloc et bit-patching) atteint une performance mesurée de **FPS = 3200.59** sur la configuration matérielle indiquée. Cette valeur provient d'un benchmark réalisé avec la commande :

```
python conway.py profile --grid-size 4000
```

5 Version Finale : Bit-Patching et Registres 64-bits

Version	FPS
Naive (Booléenne)	1001.00
Naive (Branchements)	1057.26
Shared Memory	1165.18
Lookup Table (4x4)	2455.34
Lookup Table + uint16	**3200.59**