

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 :

```
1 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) :

```
1 new_grid[idx] = 0;  
2 if (grid[idx] == 0 && alive_neighbors == 3 ||  
3     (grid[idx] == 1 && (alive_neighbors == 4 || alive_neighbors == 3)))  
4     {  
5         new_grid[idx] = 1;  
6     }  
7 else if (grid[idx] == 1 && (alive_neighbors == 4 || alive_neighbors ==  
8     3)) {  
9     new_grid[idx] = 1;  
10 }
```

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.

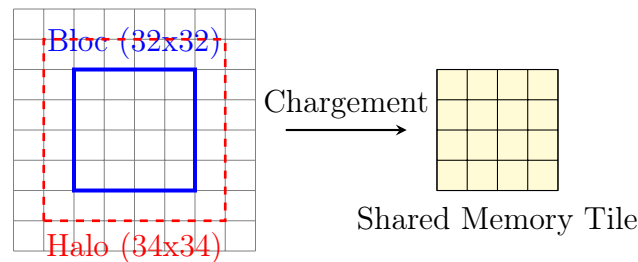


Figure 1: Schéma du chargement de la tuile et du halo en mémoire partagée.

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 :

```
1 __global__ void game_of_life_kernel(unsigned char *grid, unsigned char *  
    new_grid, int width, int height) {  
2     __shared__ unsigned char tile[34][34];  
3     int tx = threadIdx.x; int ty = threadIdx.y;  
4     int x = blockIdx.x * 32 + tx; int y = blockIdx.y * 32 + ty;  
5  
6     // Chargement collaboratif avec halo  
7     int tid = ty * 32 + tx;  
8     int block_start_x = blockIdx.x * 32 - 1;
```

```

9      int block_start_y = blockIdx.y * 32 - 1;
10
11      for (int i = tid; i < 34 * 34; i += 1024) {
12          int ly = i / 34; int lx = i % 34;
13          int gx = block_start_x + lx; int gy = block_start_y + ly;
14          unsigned char val = 0;
15          if (gx >= 0 && gx < width && gy >= 0 && gy < height)
16              val = grid[gy * width + gx];
17          tile[ly][lx] = val;
18      }
19      __syncthreads();
20
21      if (x < width && y < height) {
22          int sx = tx + 1; int sy = ty + 1;
23          int neighbors = tile[sy-1][sx-1] + tile[sy-1][sx] + tile[sy-1][
24          sx+1] +
25                          tile[sy][sx-1] + tile[sy][sx] + tile[sy][sx
26          +1] +
27                          tile[sy+1][sx-1] + tile[sy+1][sx] + tile[sy+1][
28          sx+1];
29          unsigned char current = tile[sy][sx];
30          new_grid[y * width + x] = (neighbors == 3) | (current & (
31          neighbors == 2));
32      }
33  }

```

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).

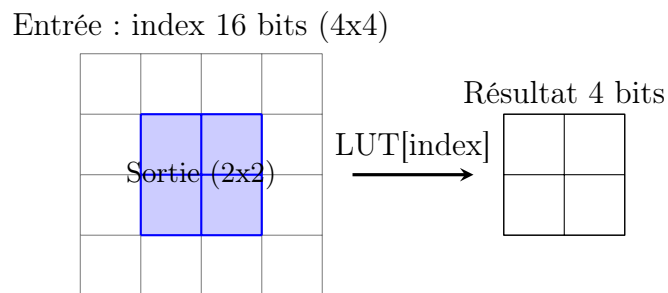


Figure 2: Principe de la LUT : Un voisinage 4x4 détermine le futur des 4 cellules centrales.

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).

1 // Extrait du kernel

```

2 int tile_r = 2 * ty;
3 int tile_c = 2 * tx;
4 uint16_t state_idx = 0;
5
6 // Construction de l'index 16 bits depuis la shared memory
7 #pragma unroll
8 for (int r = 0; r < 4; r++) {
9     for (int c = 0; c < 4; c++) {
10         if (tile[tile_r + r][tile_c + c]) {
11             state_idx |= (1 << (r * 4 + c));
12         }
13     }
14 }
15
16 // Lecture unique en memoire pour 4 pixels
17 unsigned char res = lut[state_idx];
18
19 // Ecriture des 4 pixels
20 new_grid[...] = (res >> 0) & 1;
21 new_grid[...] = (res >> 1) & 1;
22 // ... (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.

Résumé des performances :

Version	FPS
Naive (Booléenne)	1001.00
Naive (Branchements)	1057.26
Shared Memory	1165.18
Lookup Table (Precalc)	2455.34