Optimisation du Game of Life sur GPU

De l'implémentation naïve aux registres 64 bits

Hugo

16 Février 2026

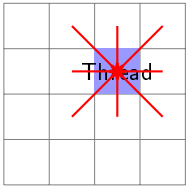
Introduction

- **Objectif :** Accélérer la simulation du Jeu de la Vie de Conway sur GPU (CUDA).
- **Plateforme :** NVIDIA GeForce GTX 1070.
- **Métrique :** Frames Per Second (FPS) sur une grille large.
- **Stratégie :** Réduire les accès mémoire et maximiser le parallélisme.

V1 & V2 : Approche Naïve (Global Memory)

Principe :

- 1 thread = 1 cellule.
- Lecture directe en mémoire globale pour chaque voisin (8 lectures/thread).
- V1 : Calcul booléen.
- V2 : Branchements (If/Else).



8 Accès Global Memory par Thread

Problème :

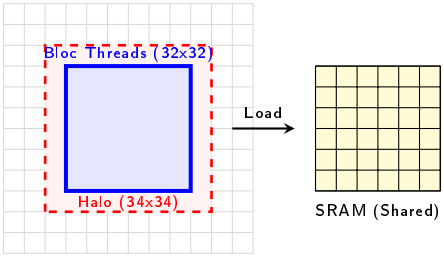
- Latence mémoire élevée.
- Redondance des lectures (chaque cellule est lue 9 fois par des threads voisins).

Performance : $\approx$ 1000 - 1050 FPS

V3 : Shared Memory & Tuiling

Optimisation :

- Chargement collaboratif d'une tuile (Tile) en mémoire partagée (rapide).
- Ajout d'un **Halo** pour gérer les bords.
- Accès Globaeuix réduits drastiquement.



Gain

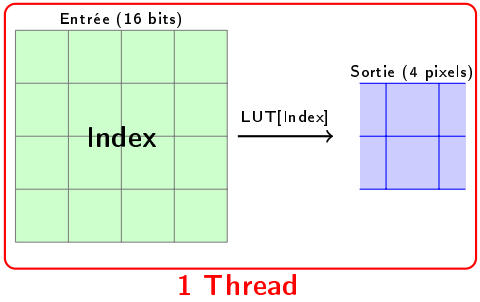
Réutilisation des données entre threads voisins via le cache L1/Shared.

Performance : 1165 FPS (+10%)

V4 : Lookup Table (LUT)

Changement de Paradigme :

- Précalculer toutes les évolutions possibles pour un petit bloc.
- Un voisinage 4×4 (16 bits) détermine l'état futur du centre 2×2.
- **Vectorisation :** 1 thread calcule 4 pixels d'un coup.
- Plus de logique conditionnelle, juste un accès tableau.



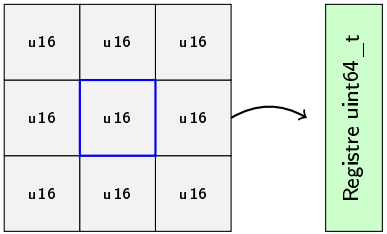
Performance : 2455 FPS (+110%)

V5 : Bit-Patching & Registres 64-bits (Le Concept)

Problematique : La V4 est rapide mais fait encore beaucoup d'accès mémoire pour construire l'index 16-bits.

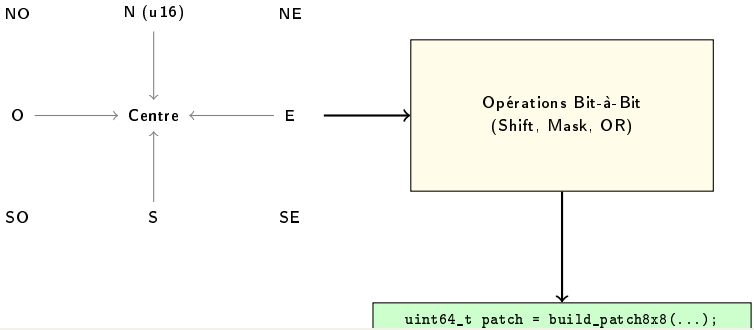
Solution "Bit-Patching" :

1. Stocker la grille sous forme d'entiers `uint16_t` (4×4 pixels).
2. Charger les 9 voisins (3×3 blocks) en registres.
3. Construire un **"Super-Patch"** virtuel de 6×6 ou 8×8 bits dans un seul registre `uint64_t`.
4. Extraire les index LUT par décalages de bits (shift).
5. **Zéro accès mémoire** pendant le calcul.



V5 : Détail du Bit-Patching (Code CUDA)

Construction du voisinage sans accès mémoire :




```
// Extrait de conway.cu
uint64_t patch = 0;
// Reconstitution ligne par ligne (bits)
// Exemple pour la partie Nord (Row 0 patch)
uint8_t mid = (N >> (4*src)) & 0xF;
uint8_t left = (NO >> (4*src+2)) & 3;
// ...
patch |= (uint64_t)row << (8*i);
```

Synthèse des Performances

Version	FPS
V1 (Naïve Bool)	1001.00
V2 (Naïve If)	1057.26
V3 (Shared)	1165.18
V4 (LUT 4x4)	2455.34
V5 (LUT u16+Reg)	3208.89

Conclusion

Résumé des gains

- **Global → Shared Ops :** Gain modéré (+10%).
- **Logique → LUT :** Gain massif par vectorisation (x2.1).
- **Shared → Registres (Bit-Patching) :** Gain final (+30%).

Performance finale :

- **x3.2** plus rapide que l'implémentation naïve.
- Utilisation optimale de la bande passante et des registres 64 bits du GPU.