

Optimisation du Game of Life sur GPU

De l'implémentation naïve au Bit-Packing

A5 TP Conway

NVIDIA RTX 3080 — Grille 32k × 32k

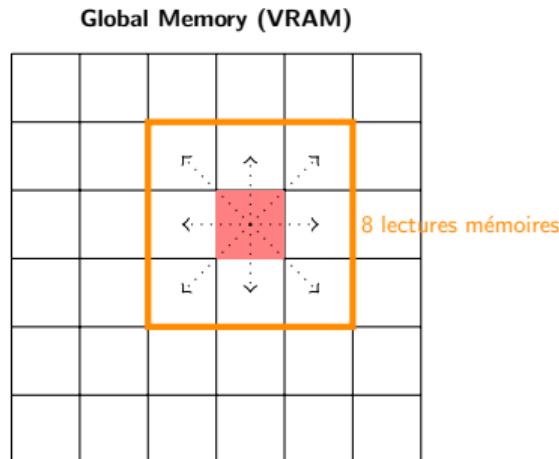
48 FPS → 416 FPS

Version 1: noif-char (Naïve)

Taille Grille: 32768×32768 octets
 ≈ 1 Go de RAM

FPS: 48.69
Type des données: char (1 octet = 1 cellule)
Pas de Shared Memory

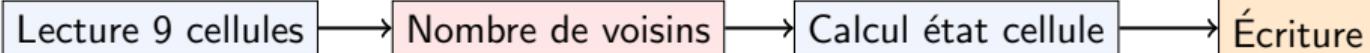
Taille bloc: 32×32 threads (calculent 32×32 cellules)



Logique “Branchless” (noif)

```
// On compte les voisins vivants
int neighbors = 0;
for (int dy = -1; dy < 2; dy++) {
    for (int dx = -1; dx < 2; dx++) {
        if (x + dx >= 0 && x + dx < width && y + dy >= 0 && y + dy < height && (dy != 0 || dx != 0)) {
            neighbors += grid[idx + dx + dy * width];
        }
    }
}

// Calcul conditionnel sans 'if'
int alive = (neighbors == 3) | (grid[y * width + x] == 1 && neighbors == 2);
new_grid[idx] = alive;
```

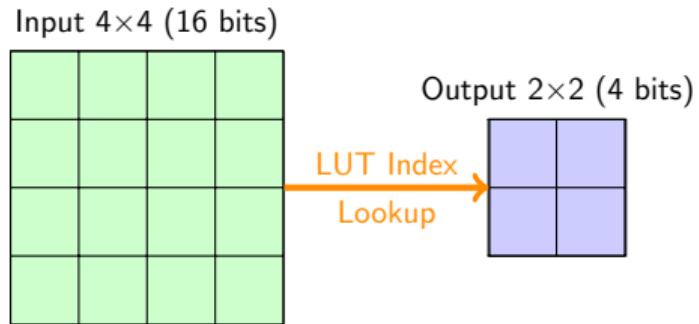


Version 2: Lookup Table 4×4

Performance:

- **FPS:** 112.76 ($\times 2.3$)
- **Stratégie:** Pré-calcul (LUT)
- **Mémoire:** Utilisation de la **Shared Memory**

Concept: Réduire les calculs (comptage voisins) par une simple lecture tableau.



```
unsigned char host_lut[65536]
```

Construction de l'index LUT

```
// Construction de l'index 16-bits
uint16_t state_idx = 0;
#pragma unroll
for (int r = 0; r < 4; r++) {
    #pragma unroll
    for (int c = 0; c < 4; c++) {
        if (tile[tile_r + r][tile_c + c]) {
            state_idx |= (1 << (r * 4 + c));
        }
    }
}
unsigned char res = lut[state_idx];
```

b30	b31	b32	b33
b20	b21	b22	b23
b10	b11	b12	b13
b00	b01	b02	b03

→ uint16_t state_idx

Bloc 4×4 (voisinage)

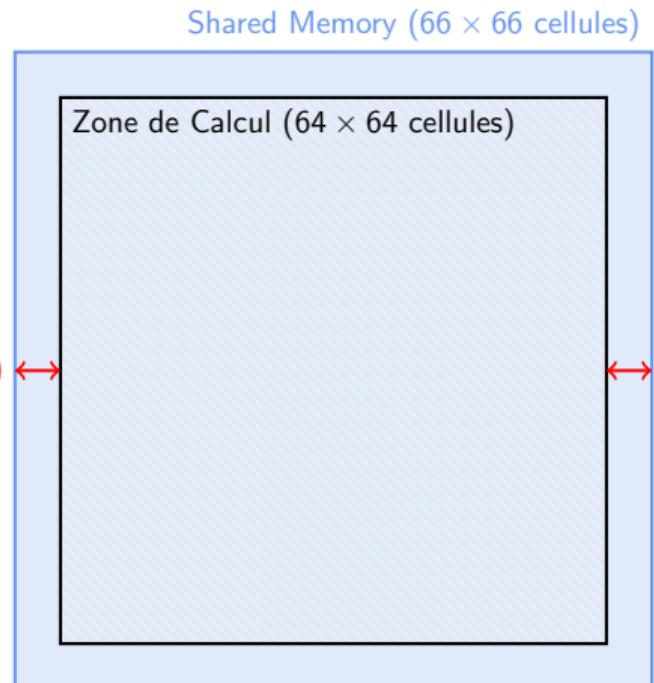
Gestion Mémoire: Stratégie par Bloc

Un Thread calcule **2×2 cellules**

Bloc 32×32 threads

Surface calculée: **64×64 cellules**

Shared Memory: **66×66 cellules**



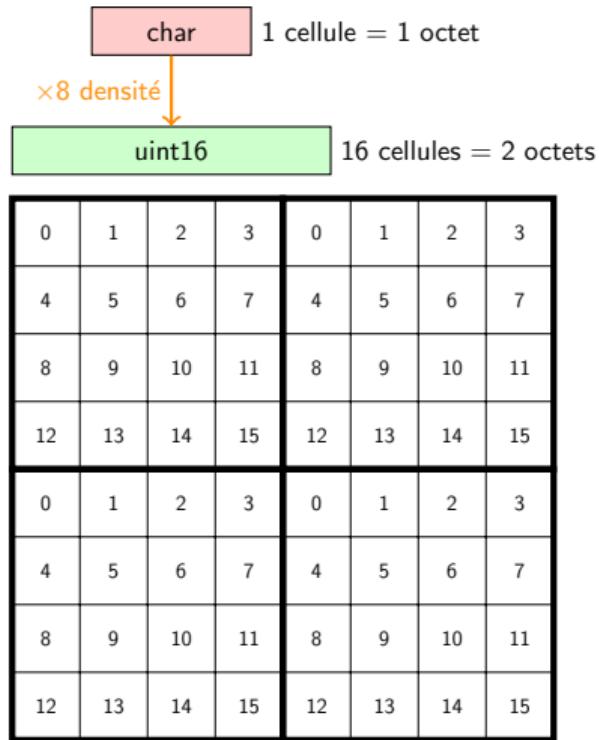
Version 3: uint16 (Bit-Packing)

Performance:

- **FPS: 416.51** ($\times 8.5$ vs Naïve)
- **Stratégie:** Compression Mémoire

Principe:

- 1 cellule = 1 bit (pas 1 octet)
- Grille 32768×32768 cellules
- $\frac{32768 \times 32768}{8 \text{ bits/octet}} = 128 \text{ Mo}$



Architecture du Kernel uint16

$1 \text{ uint16} = 1 \text{ thread}$

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

Architecture du Kernel uint16

$1 \text{ uint16} = 1 \text{ thread}$

Utilisation de la LUT 4×4

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

Architecture du Kernel uint16

Utilisation de la LUT 4×4
Construction de l'indice 16-bits
à partir des uint16 voisins

$1 \text{ uint16} = 1 \text{ thread}$

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

Architecture du Kernel uint16

Utilisation de la LUT 4×4

Construction de l'indice 16-bits
à partir des uint16 voisins

Patch suivant

$1 \text{ uint16} = 1 \text{ thread}$

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

Architecture du Kernel uint16

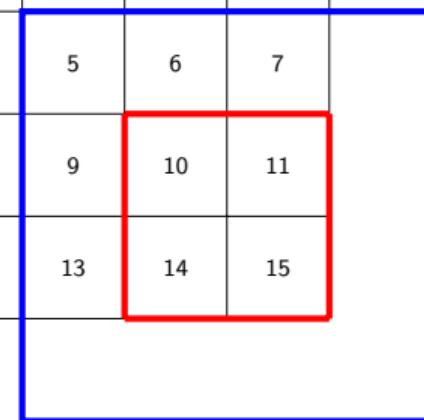
1 uint16 = 1 thread

Utilisation de la LUT 4×4

Construction de l'indice 16-bits
à partir des uint16 voisins

Patch suivant

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



The diagram shows a 4x4 grid of indices from 0 to 15. A blue rectangle highlights the top-left 3x3 subgrid (indices 0-8), which corresponds to the 'Patch suivant'. A red rectangle highlights the bottom-right 2x2 subgrid (indices 10-15), which represents the neighborhood used to construct the 16-bit index.

Architecture du Kernel uint16

1 uint16 = 1 thread

Utilisation de la LUT 4×4

Construction de l'indice 16-bits
à partir des uint16 voisins

Patch suivant

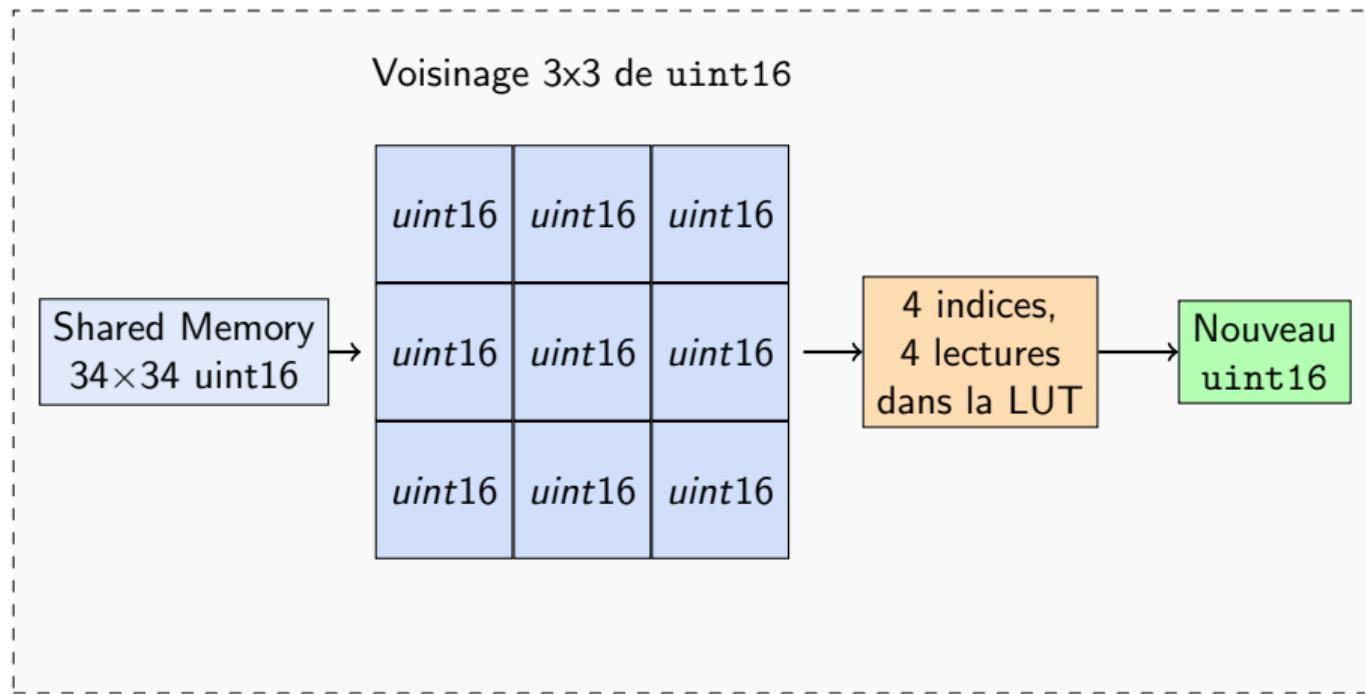
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The diagram illustrates a 4x4 grid of indices, indexed from 0 to 15. A large blue rectangle encloses the entire grid. A smaller red rectangle highlights a 3x3 subgrid in the center, containing indices 8, 9, 10, 12, 13, and 14. This visual representation demonstrates how a larger patch is processed using a smaller LUT (Look-Up Table) of size 4x4.

Architecture du Kernel uint16

1 Thread = uint16

Bloc 32×32 threads = 128×128 cellules



Résumé des Gains de Performance

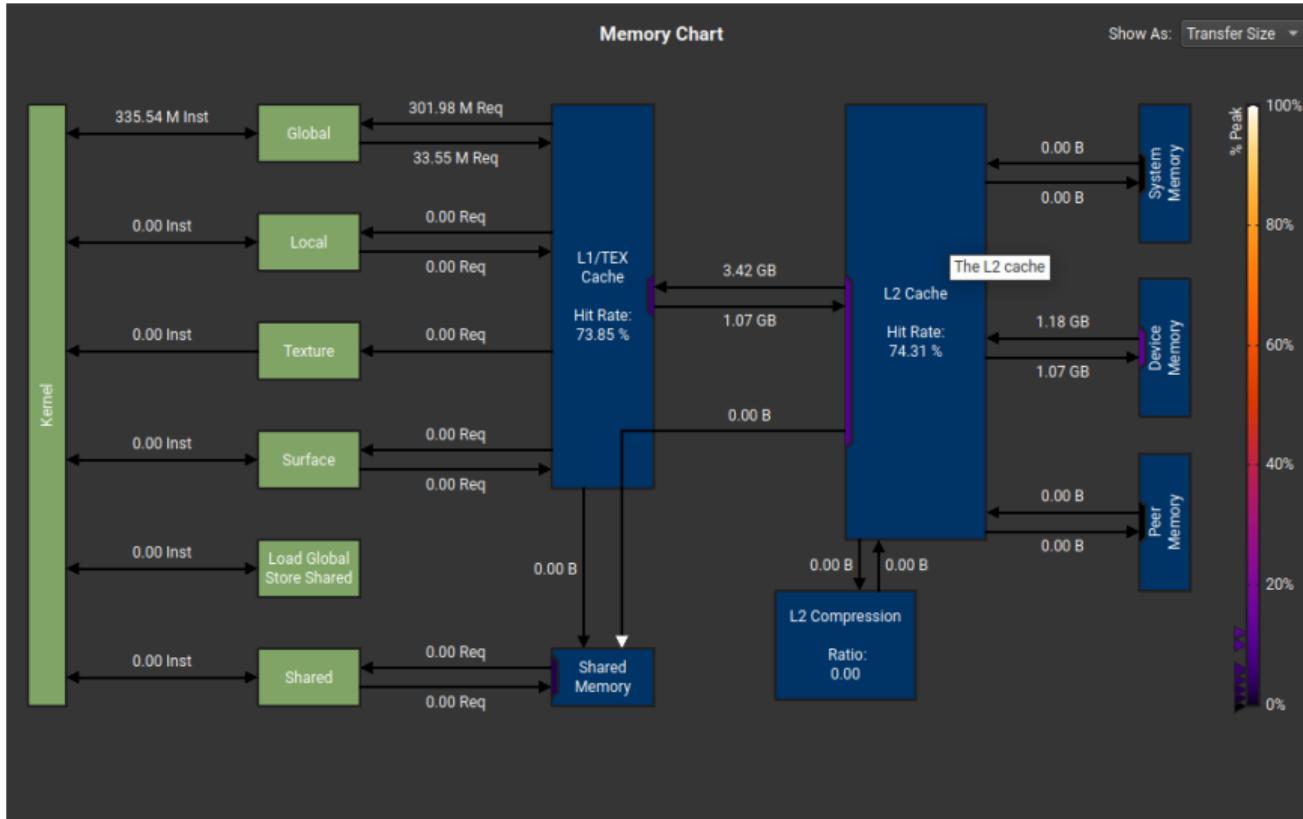
Métriques de Débit et Temps d'Exécution

Version	Thread/cellule	Durée (ms)	BP Mém. (Go/s)	BP Mém. (%)	FPS
noif-char	1	25.21	89.42	32.72	48.81
lookup4×4	4	11.21	196.02	48.30	152.04
uint16	16	3.03	90.54	25.85	416.51

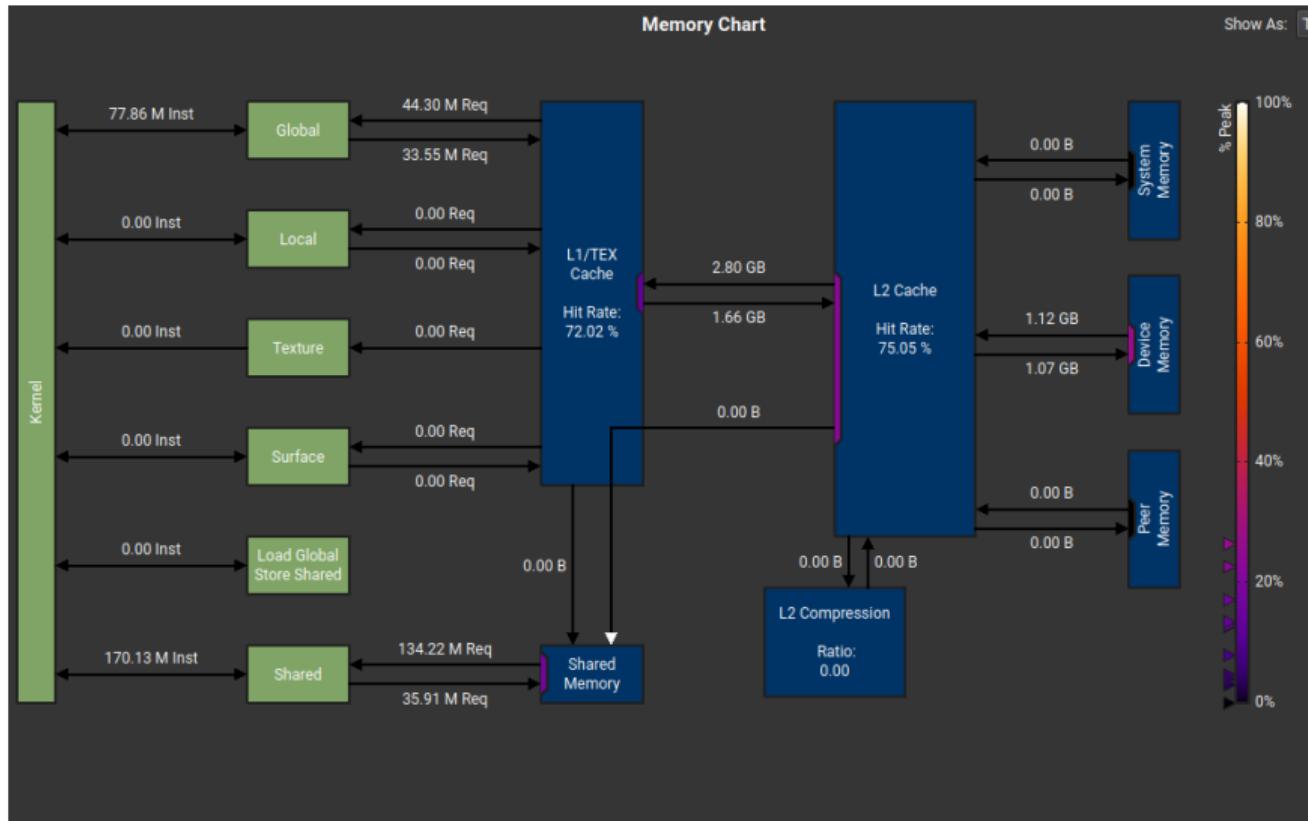
Efficacité Matérielle

Version	Hit L1/TEX (%)	Occupation Mém. (%)	SM Busy (%)	Calcul (%)
noif-char	73.85	16.56	32.46	32.72
lookup4×4	72.02	31.05	46.56	48.30
uint16	95.56	20.68	39.18	37.25

Profiling: noif-char



Profiling: lookup4×4



Profiling: uint16 (Division par 8 des accès mémoire)

