Programmation des architectures Parallèles

Lucas Marques, Matis Duval

21 avril 2025

Rapport Projet : Le jeu de la vie - Troisième Jalon

Résumé

Ce troisième jalon du projet se concentre sur l'optimisation du Jeu de la Vie à travers deux axes principaux : la vectorisation des calculs pour réduire l'empreinte mémoire et améliorer les performances en travaillant sur plusieurs cellules à la fois, et l'implémentation d'une version GPU utilisant OpenCL. Nous présentons nos approches pour exploiter les instructions SIMD et les architectures massivement parallèles des cartes graphiques, qui ont certaines caractéristiques en commun.

1 Introduction

Après avoir optimisé les calculs séquentiels et parallèles sur CPU dans les jalons précédents, nous nous attaquons maintenant à deux nouvelles dimensions d'optimisation : la vectorisation des calculs via les instructions SIMD (AVX2/AVX512) et le portage sur architecture GPU. Ces optimisations visent à exploiter pleinement les capacités des machines modernes, tant au niveau des unités vectorielles des CPUs que du parallélisme massif des GPUs.

2 AVX2 et AVX512

2.1 Expérimentation avec AVX2

2.1.1 Version naïve

Nous avons commencé par travailler sur une version qui ne faisait que 3 loads, et décalait ensuite les indices dans des copies de nos 3 vecteurs, mais nous n'avons pas réussi à faire fonctionner cette version qui présentait quelques effets de bords. Cette version (toujours disponible dans le code sous le nom de life_do_tile_firstidea) finissait toujours par avoir un nombre de voisins faux pour certaines cellules. Après en avoir discutés rapidement en cours, nous avons décidés d'abandonner cette version. Si vous regardez son code vous constaterez que l'on revient en séquentiel à un moment. C'est car nous avons prit pour habitude de partir d'une version séquentielle pour ensuite la vectoriser petit-à-petit ce qui nous aidait à trouver les bugs plus facilement, plutôt que de devoir vérifier chaque étape une par une si l'on fait tout d'un coup. Nous avons alors implémentés une version naïve assez classique où l'on vectorise de la manière la plus simple possible notre jeu de la vie en traitant des lignes de cellules (32 cellules par vecteur). 9 loads sont effectués à chaque itération.

```
#define M256I_LOADU(y, x)                                                      \
  _mm256_loadu_si256 ((const __m256i *)table_cell (_table, y, x))

// creates a vector containing number of neighbors in a vector
static inline __m256i _mm256_compute_neighbors (
    __m256i vec_top_shift_left, __m256i vec_cell_shift_left,
    __m256i vec_bot_shift_left, __m256i vec_top, __m256i vec_cell,
    __m256i vec_bot, __m256i vec_top_shift_right, __m256i vec_cell_shift_right,
    __m256i vec_bot_shift_right)
{
  __m256i vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_top_shift_left, vec_cell_shift_left);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_bot_shift_left);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_top);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_cell);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_bot);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_top_shift_right);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_cell_shift_right);
  vec_cell_line_neigh_count =
      _mm256_add_epi8 (vec_cell_line_neigh_count, vec_bot_shift_right);

  // we substract ourself from this count
  vec_cell_line_neigh_count =
      _mm256_sub_epi8 (vec_cell_line_neigh_count, vec_cell);
  return vec_cell_line_neigh_count;
}

// do the game of life rules from a neighbor vector and updates the next table
static inline __m256i
_mm256_compute_cells (__m256i vec_cell_line_neigh_count, __m256i vec_cell,
                      __m256i only_threes, __m256i only_twos, __m256i only_ones,
                      __m256i only_zeros)
{
  __m256i three_neighbors =
      _mm256_cmpeq_epi8 (vec_cell_line_neigh_count, only_threes);

  __m256i two_neighbors =
      _mm256_cmpeq_epi8 (vec_cell_line_neigh_count, only_twos);

  __m256i alive_mask = _mm256_cmpgt_epi8 (vec_cell, only_zeros);

  __m256i two_neighbors_and_alive =
      _mm256_and_si256 (two_neighbors, alive_mask);

  __m256i next_alive_mask =
      _mm256_or_si256 (three_neighbors, two_neighbors_and_alive);

  __m256i next_alive = _mm256_and_si256 (next_alive_mask, only_ones);
  return next_alive;
}

static inline char compute_from_vects (
    __m256i vec_top_shift_left, __m256i vec_cell_shift_left,
    __m256i vec_bot_shift_left, __m256i vec_top, __m256i vec_cell,
    __m256i vec_bot, __m256i vec_top_shift_right, __m256i vec_cell_shift_right,
    __m256i vec_bot_shift_right, int i, int j, __m256i only_threes,
    __m256i only_twos, __m256i only_ones, __m256i only_zeros)
{
  // then we compute the neighbor count
  __m256i vec_cell_line_neigh_count = _mm256_compute_neighbors (
      vec_top_shift_left, vec_cell_shift_left, vec_bot_shift_left, vec_top,
      vec_cell, vec_bot, vec_top_shift_right, vec_cell_shift_right,
      vec_bot_shift_right);
  // we can now apply rules
  __m256i next_alive =
      _mm256_compute_cells (vec_cell_line_neigh_count, vec_cell, only_threes,
                            only_twos, only_ones, only_zeros);
  // store the result
  _mm256_storeu_si256 ((__m256i *)table_cell (_alternate_table, i, j),
                       next_alive);

  // and finally compute for changes
  __m256i diff = _mm256_xor_si256 (vec_cell, next_alive);
  return !_mm256_testz_si256 (diff, diff);
}

int life_do_tile_avx2 (const int x, const int y, const int width,
                       const int height)
{
  if (x < 32 || x + width >= DIM - 33) {
    return life_do_tile_opt (x, y, width, height);
```

```
  }
  char change = 0;

  int x_start = x;
  int x_end   = x + width;
  int y_start = (y == 0) ? 1 : y;
  int y_end   = (y + height >= DIM) ? DIM - 1 : y + height;

  // some constants we're gonna use
  __m256i only_threes = _mm256_set1_epi8 (3);
  __m256i only_twos   = _mm256_set1_epi8 (2);
  __m256i only_ones   = _mm256_set1_epi8 (1);
  __m256i only_zeros  = _mm256_setzero_si256 ();

  for (int i = y_start; i < y_end; i++) {
    for (int j = x_start; j < x_end; j += 32) {
      __m256i vec_top_shift_left  = M256I_LOADU (i - 1, j - 1);
      __m256i vec_cell_shift_left = M256I_LOADU (i, j - 1);
      __m256i vec_bot_shift_left  = M256I_LOADU (i + 1, j - 1);

      __m256i vec_top  = M256I_LOADU (i - 1, j);
      __m256i vec_cell = M256I_LOADU (i, j);
      __m256i vec_bot  = M256I_LOADU (i + 1, j);

      __m256i vec_top_shift_right  = M256I_LOADU (i - 1, j + 1);
      __m256i vec_cell_shift_right = M256I_LOADU (i, j + 1);
      __m256i vec_bot_shift_right  = M256I_LOADU (i + 1, j + 1);

      change |= compute_from_vects (
          vec_top_shift_left, vec_cell_shift_left, vec_bot_shift_left, vec_top,
          vec_cell, vec_bot, vec_top_shift_right, vec_cell_shift_right,
          vec_bot_shift_right, i, j, only_threes, only_twos, only_ones,
          only_zeros);
    }
  }
  return change;
}
```

Nous n'allons pas effectuer de comparaison sur une version séquentielle étant donné que comme dit précédemment, elle revient à appeler do_tile_opt car elle n'est pas tuilée. Nous utiliserons la version tilled qui - bien que plus lente qu'une version séquentielle - nous permettra de comparer "à armes égales", en utilisant également une taille de tuile non optimisée mais suffisament petite pour forcer la version AVX2 à faire la majorité du travail. Nous nous attendons à voir de gros speedups entre default et avx2, mais étant donné que la version opt était déjà bien vectorisable, les speedups avec cette dernière devraient être moins impressionnants.

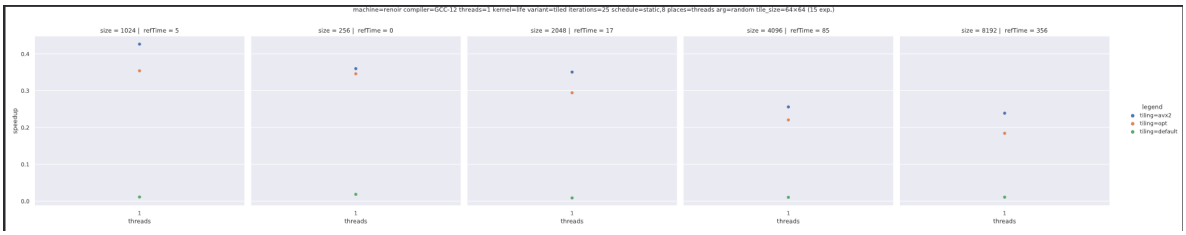

FIGURE 1 – Comparaison de nos trois tuilages

Nous constatons donc bel et bien un important speedup entre la version default et notre version AVX2, de l'ordre de x15/x20, mais bien moins important entre les versions opt et AVX2. On voit cependant que la vectorisation à la main est plus efficace que la vectorisation automatique du compileur. C'est assez cohérent étant donné que notre usage des intrinsèques AVX est adapté à nos règles, là où le compileur semblait lors de notre dernier rapport se contentait d'optimiser la partie calcul du nombre de voisins.

2.1.2 Optimisation des paramètres

Nous allons donc commencer par chercher un nombre de threads optimal pour notre version AVX2, car nous savons que lorsque trop de coeurs exécutent du code AVX en même temps, le CPU a tendance à réduire la fréquence de ces coeurs, ce qui impact les performances globales.

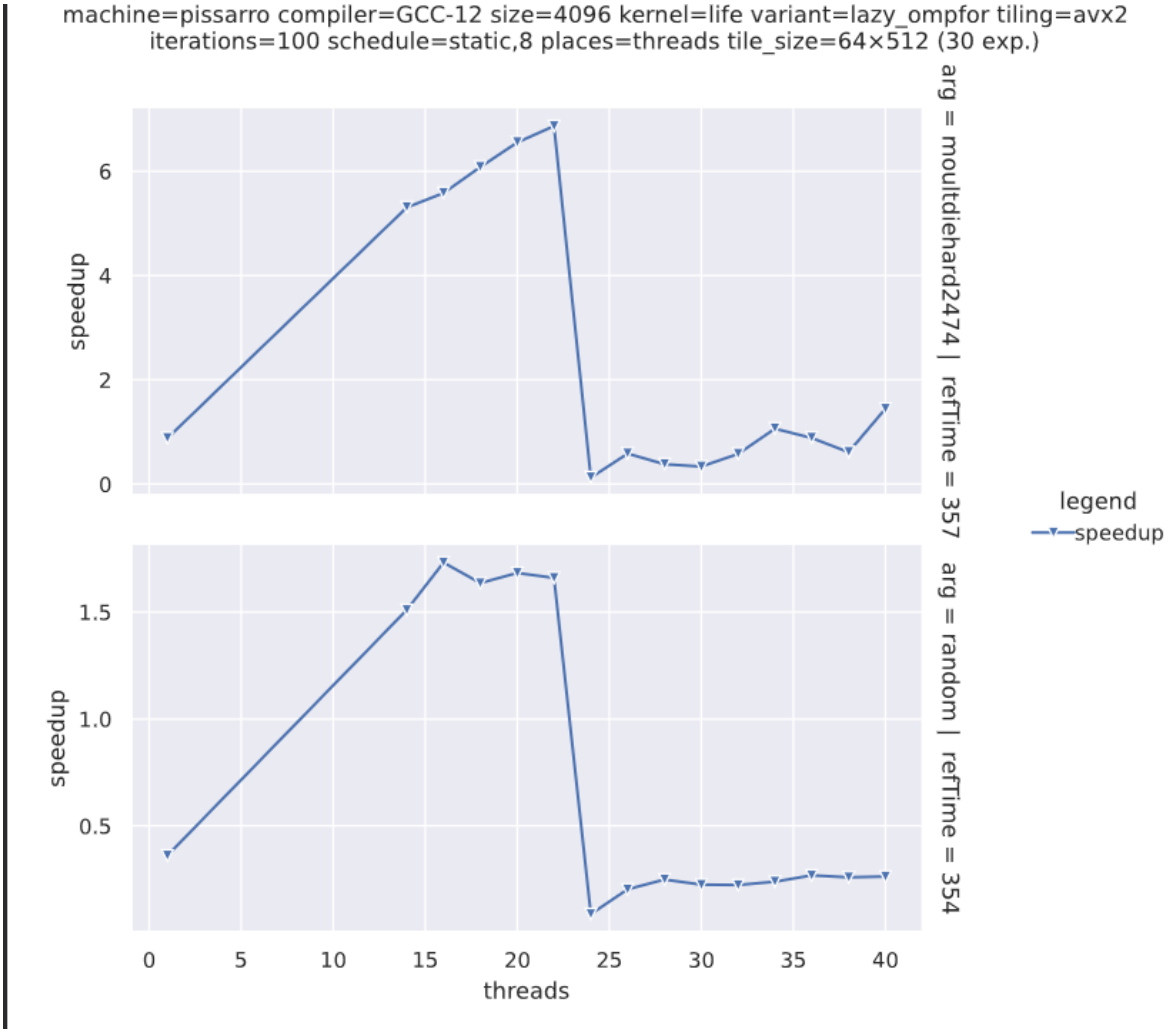

FIGURE 2 – Speedups en fonction du nombre de threads sur pissarro, version AVX2

Voici le premier graph que nous avons générés. Ce dernier nous a tout de suite interloqué, et s'en est suivi une demi-heure d'expérimentations en tout genre pour comprendre ce qu'il se passait... C'est le nombre de threads à partir duquel les performances s'effondrent qui nous a fait réaliser notre erreur. Notre première pensée a été de se dire que le placement des threads et de la mémoire n'était pas bon, mais après un lscpu nous nous sommes rappelés que Pissarro n'avait pas la même architecture que les autres, et avait deux fois moins de coeurs... Repartons donc de zéro !

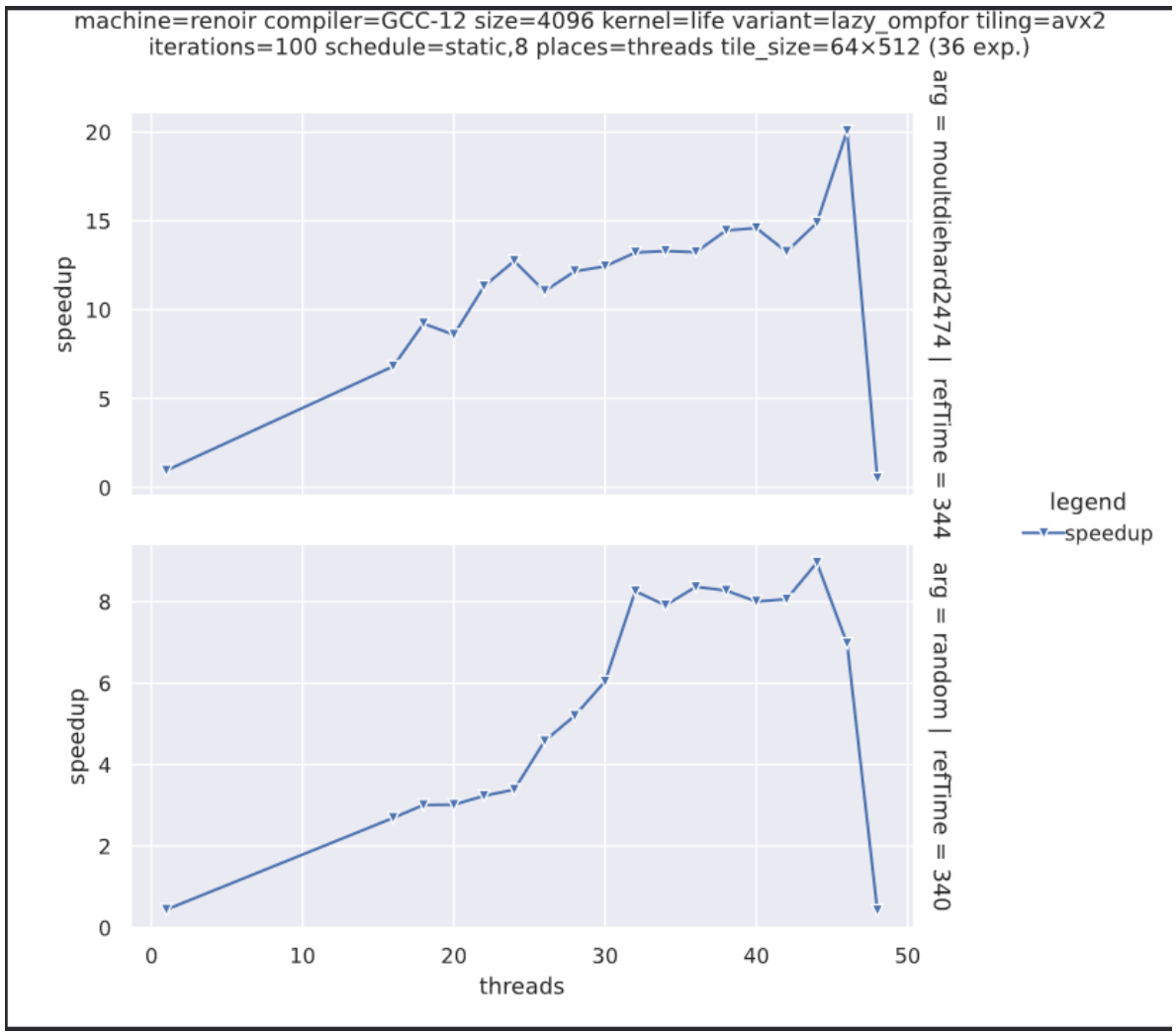

FIGURE 3 – Speedups en fonction du nombre de threads, version AVX2

On constate déjà des comportements plus... rationnels. Le nombre optimal de threads pour notre code semble donc se situer entre 44 et 46. Ce nombre était plus faible pour nos précédentes versions (aux alentours de 36 threads). On semble donc constater que l'on tire un meilleur parti des threads avec cette version AVX2.

2.1.3 Recherche d'une taille de tuiles optimale

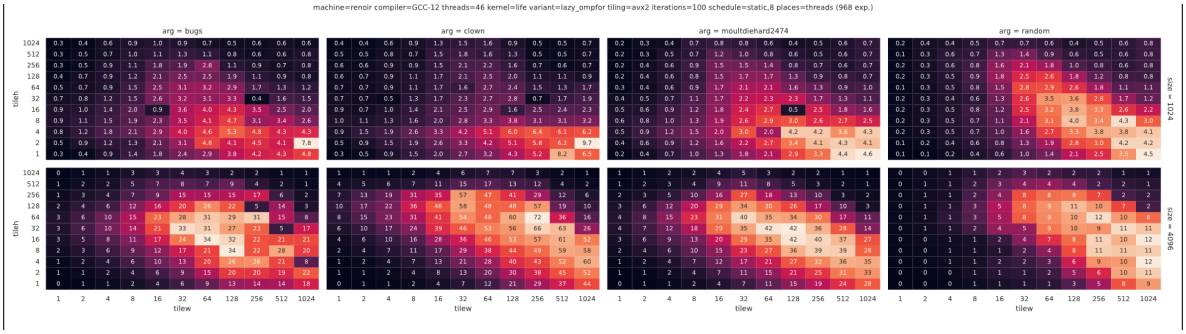

FIGURE 4 – Speedup en fonction de la taille de tuile, AVX2 + version lazy OpenMP

Nous avions déjà une version qui favorisait le chargement de tuiles très larges, donc les résultats obtenus ne diffèrent pas pas beaucoup de notre précédente version. C'est logique étant donné qu'on traite plus de colonnes en largeur, on aurait pu s'attendre à avoir un spot dans la heatmap encore plus décalé vers la droite (tuiles larges) étant donné que l'on traite 32 tuiles à la fois. L'inverse aurait été étonnant !

2.1.4 Apparté sur le scheduling

M. Namyst avait suggéré en cours de jouer avec les tailles de blocs dans nos distributions de charge aux threads, nous avons donc décidés d'appliquer ce conseil. Pour cela, nous utilisons simplement ce code pour nos expériences :

```
ompenv["OMP_SCHEDULE"] = [f"static,{i}" for i in range(1, 33, 1)] + [
    f"dynamic,{i}" for i in range(1, 33, 1)
]
```

Nous faisons déjà quelques tests pour enlever les valeurs inutiles afin de ne pas surcharger le graph du rapport. On constate alors quelques comportements.

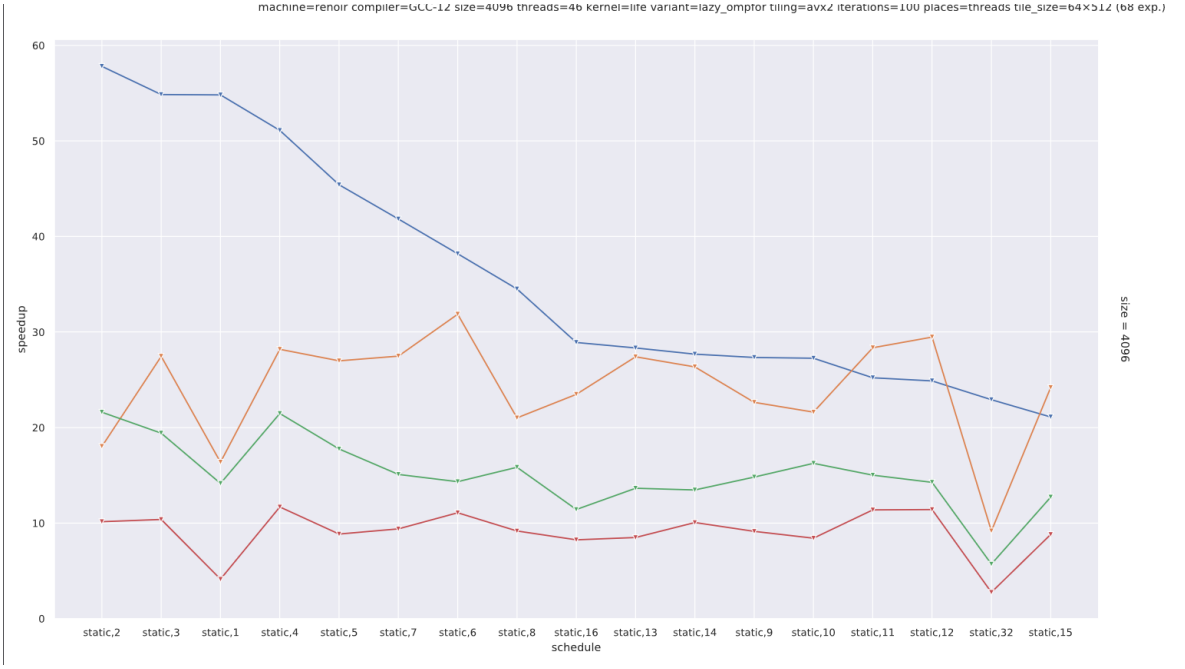

FIGURE 5 – Speedups en fonction du scheduling

Le presque facteur 60 n'est à prime abord pas très intéressant, on sait que Clown est très efficace en version lazy, mais on peut quand même se pencher rapidement sur ce speedup. Comparons la distribution des threads du meilleur et du pire cas :

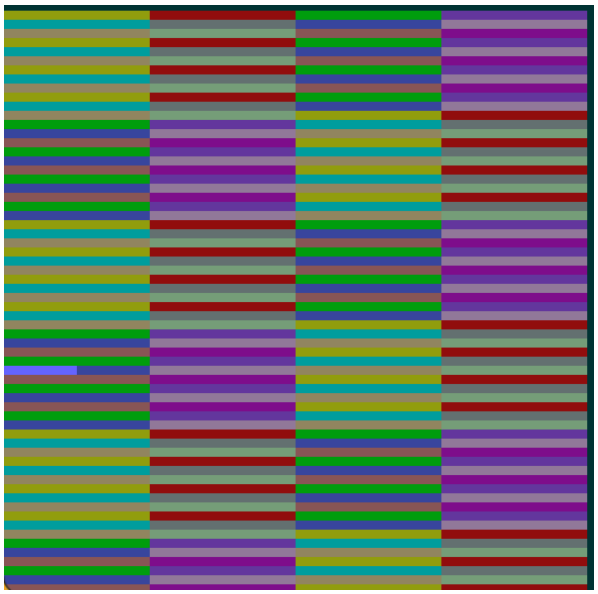

FIGURE 6 – Distribution des tuiles pour static,2

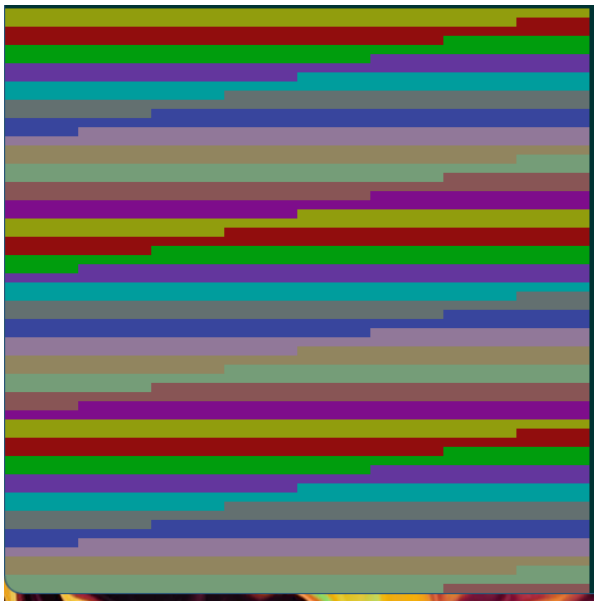

FIGURE 7 – Distribution des tuiles pour static,15

L'alignement induit par la distribution *static*, 15 fait que souvent, le même thread va se retrouver à calculer les tuiles en (x, y), $(x+1, y)$, $(x+2, y)$... Or, dans une configuration à faible entropie comme clown, si les threads ne "se marchent pas dessus", étant donné que l'espace de travail est proche et peut être couvert par peu de threads, on réparti tout simplement mieux la charge. On peut donc en déduire que pour optimiser au mieu le paramètre schedule, il ne faut pas simplement utiliser une "valeur magique" mais plutôt réfléchir en fonction de la taille des tuiles et de la distribution de notre problème, ainsi que de la distribution de son entropie pour les version lazy.

2.1.5 Suite de Développement AVX2

Nous n'avons à ce jour pas trouvé de solution pour gérer les bords sur des petites images exécutées en séquentiel. En effet, nous repassons le calcul des tuiles qui dépasseraient des vecteurs à notre fonction **do_tile_opt**, et toute tentative de mitiger cela en faisant tout de même une partie du calcul en vectoriel en introduisant des conditions dans la boucle se sont révélées bien plus lentes. C'est pour cela que par exemple sur une variante séquentielle, étant donné que nous ne calculons qu'une seule tuile, nous sommes plus lent en AVX qu'avec notre version optimisée précédemment développée.

Nous constatons de réelles accélérations sur des grosses images avec la version lazy. Il est sûrement possible de faire fonctionner le tout avec du masquage, mais nous n'avons pas trouvé de solution fonctionnelle.

Nous avons également tenté une version avec moins de loads en nous inspirant de ce dont nous avions discuté en cours. Nous avons trouvé une solution pour le faire verticalement et ne recharger les 9 vecteurs que pour chaque colonne, mais cette version s'exécute pas substantiellement plus rapidement. Elle n'est probablement pas très cache-friendly cependant, étant donné qu'elle s'exécute en ligne. Nous n'avons pas trouvés une solution pour le faire en colonnes. Nous avons pensés à effectuer une sorte de traversée en diagonale (au lieu de cette version en ligne verticale) mais n'avons pas réussi à l'implémenter.

```
int life_do_tile_avx2_lessload (const int x, const int y, const int width,
                                const int height)
{
  if (x < 32 || x + width >= DIM - 33) {
    return life_do_tile_opt (x, y, width, height);
  }
  char change          = 0;
  int x_start          = (x == 0) ? 1 : x;
  int x_end            = (x + width >= DIM) ? DIM - 1 : x + width;
  int y_start          = (y == 0) ? 1 : y;
  int y_end            = (y + height >= DIM) ? DIM - 1 : y + height;
  __m256i only_threes = _mm256_set1_epi8 (3);
  __m256i only_twos   = _mm256_set1_epi8 (2);
  __m256i only_ones   = _mm256_set1_epi8 (1);
  __m256i only_zeros  = _mm256_setzero_si256 ();

  for (int j = x_start; j < x_end; j += 32) {
    __m256i vec_top_shift_left  = M256I_LOADU (y_start - 1, j - 1);
    __m256i vec_top             = M256I_LOADU (y_start - 1, j);
    __m256i vec_top_shift_right = M256I_LOADU (y_start - 1, j + 1);

    __m256i vec_cell_shift_left  = M256I_LOADU (y_start, j - 1);
    __m256i vec_cell             = M256I_LOADU (y_start, j);
    __m256i vec_cell_shift_right = M256I_LOADU (y_start, j + 1);

    for (int i = y_start; i < y_end; i++) {
      __m256i vec_bot_shift_left  = M256I_LOADU (i + 1, j - 1);
      __m256i vec_bot             = M256I_LOADU (i + 1, j);
      __m256i vec_bot_shift_right = M256I_LOADU (i + 1, j + 1);

      change |= compute_from_vects (
          vec_top_shift_left, vec_cell_shift_left, vec_bot_shift_left, vec_top,
          vec_cell, vec_bot, vec_top_shift_right, vec_cell_shift_right,
          vec_bot_shift_right, i, j, only_threes, only_twos, only_ones,
          only_zeros);

      vec_top_shift_left  = vec_cell_shift_left;
      vec_top             = vec_cell;
      vec_top_shift_right = vec_cell_shift_right;

      vec_cell_shift_left  = vec_bot_shift_left;
      vec_cell             = vec_bot;
      vec_cell_shift_right = vec_bot_shift_right;
    }
  }
  return change;
}
```

On réitère notre expérience sur une version tiled, en enlevant cette fois-ci la version default.

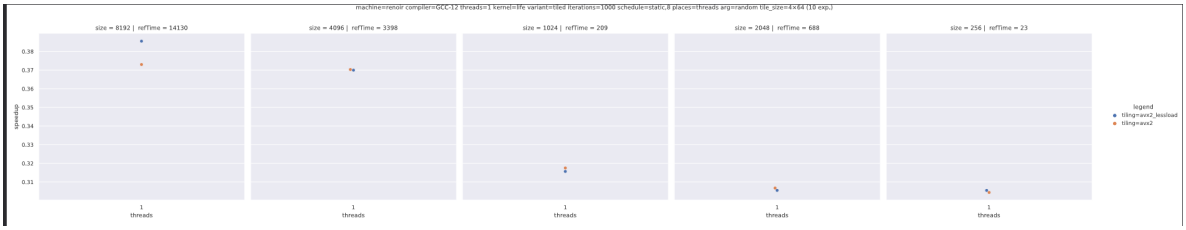

FIGURE 8 – Comparaison des performances : AVX2 standard vs AVX2 avec réduction des loads

Et on constate de légers speedups sur les tableaux de grande taille.
En jouant un peu avec la hauteur des tuiles, nous constatons cependant qu'il est possible d'améliorer les performances de la version avec moins de load, mais en poussant les comparaisons, même si pour deux tuilages équivalents il est possible de faire "gagner" la deuxième version, les paramètres optimisés pour une configuration font toujours de la version AVX2 de base la plus rapide.

2.2 Version AVX512

Cette version est une adaptation de la version AVX2 avec les fonctions appropriées pour AVX512, la seule légère différence étant au niveau du masquage utilisé pour appliquer les règles.

```
    ...
  __m512i only_threes = _mm512_set1_epi8 (3);
  __m512i only_twos   = _mm512_set1_epi8 (2);
  __m512i only_ones   = _mm512_set1_epi8 (1);
  __m512i only_zeros  = _mm512_setzero_si512 ();

  for (int i = y_start; i < y_end; i++) {
    for (int j = x_start; j < x_end; j += 64) {
      // first we load the vectors
      __m512i vec_top_shift_left = _mm512_loadu_si512 (
          (const __m512i *)table_cell (_table, i - 1, j - 1));
      __m512i vec_cell_shift_left =
          _mm512_loadu_si512 ((const __m512i *)table_cell (_table, i, j - 1));
      __m512i vec_bot_shift_left = _mm512_loadu_si512 (
          (const __m512i *)table_cell (_table, i + 1, j - 1));

      __m512i vec_top =
          _mm512_loadu_si512 ((const __m512i *)table_cell (_table, i - 1, j));
      __m512i vec_cell =
          _mm512_loadu_si512 ((const __m512i *)table_cell (_table, i, j));
      __m512i vec_bot =
          _mm512_loadu_si512 ((const __m512i *)table_cell (_table, i + 1, j));

      __m512i vec_top_shift_right = _mm512_loadu_si512 (
          (const __m512i *)table_cell (_table, i - 1, j + 1));
      __m512i vec_cell_shift_right =
          _mm512_loadu_si512 ((const __m512i *)table_cell (_table, i, j + 1));
      __m512i vec_bot_shift_right = _mm512_loadu_si512 (
          (const __m512i *)table_cell (_table, i + 1, j + 1));
     ...
```

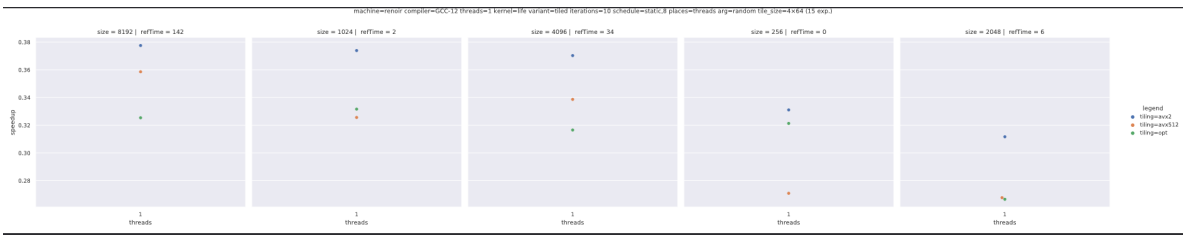

FIGURE 9 – Comparaison des performances entre opt, AVX2 et AVX512

Ce que nous remarquons nous semble contreintuitif. La version AVX512 est systématiquement plus lente que la version AVX2, parfois même que la version opt. Nous n'avons pas l'impression d'avoir de problème au niveau du code étant donné qu'il s'agit quasiment du même entre nos deux versions vectorisées, même si une erreur de notre part reste évidemment possible.
Nous avons d'abord pensés au dynamic frequency scaling, mais ce problème ne devrait dans ce cas

pas se manifester sur la version tiled qui tourne sur un seul coeur. Une autre possibilité serait que nous sommes limités par les accès mémoire, et que les accès mémoire sur des vecteurs de 512bits sont suffisament plus longs que sur 256bits pour cacher le speedup induit par le traitement de 2x plus de cellules.
Mais la cause la plus probable, où du moins la plus importante, reste le retour en mode do_tile_opt. En effet, si l'on rend le calcul des bords à la version AVX (et donc que l'on se retrouve avec des valeurs fausses pour la majorité des cas...) on obtient en effet ce résultat :

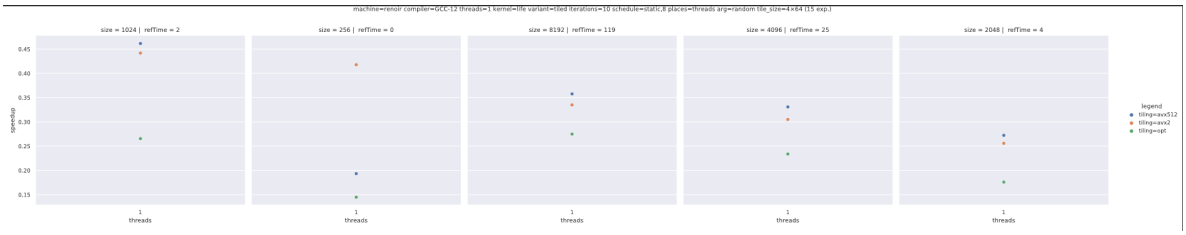

FIGURE 10 – Comparaison avec mauvais calcul des bords

Et ce dernier correspond déjà un peu plus à nos attentes...

Quand on compare deux traces, on voit également que lorsqu'on arrive sur une tâche d'un bord, on prend plus de temps (ce qui est parfaitement logique ! mais explique également une bonne partie du problème selon nous)

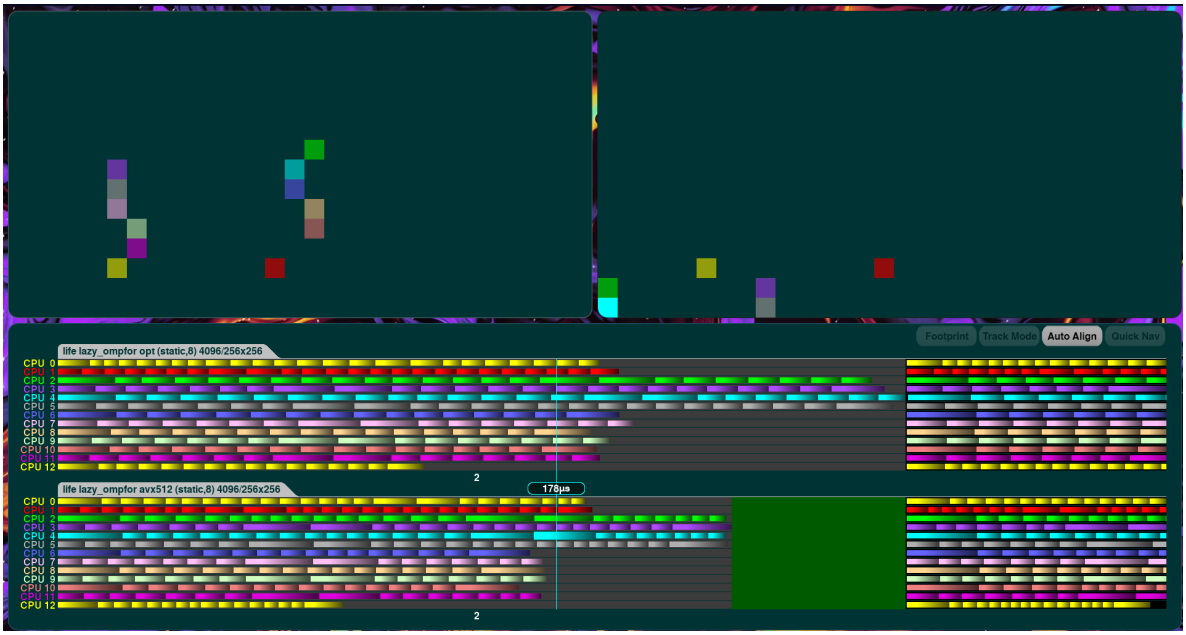

FIGURE 11 – En bleu à droite, une tuile en bordure prend bien plus de temps à calculer

On voit d'ailleurs bien sur cette trace qu'il n'y a pas une grande différence sur le temps de calcul d'une tuile, même si visuellement avx512 semble aller légèrement plus vite.

2.3 Comparaisons opt/avx2/avx512

FIGURE 12 – Comparaison entre do_tile_opt et les versions vectorisées

Comme dit précédemment, il n'y a que peu de différences entre les trois versions. Nous n'avons pas vraiment d'explication concrète à part la partie retour au scalaire de notre code, mais elle ne devrait pas selon nous tant se manifester sur des grandes tailles. Peut-être une surcharge de la bande passante vers la mémoire étant donné que l'on fait surtout des load. Nous n'avons pas trouvé comment lancer easypap avec l'option -pc pour investiguer plus loin.

2.4 Frequency Scaling

Les versions implémentées de vectorisations peuvent, comme dit précédemment, être limitées par le processeur qui descend la fréquence de certains coeurs exécutant des instructions vectorielles afin d'éviter qu'ils ne surchauffent. Nous pouvons nous appuyer sur le schéma vu en cours pour avoir une idée des performances des cœurs des machines de la salle 008.

[Modify Frequency Info]

Mode	Base	Turbo Frequency/Active Cores													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Normal	2,200 MHz	3,200 MHz	3,200 MHz	3,000 MHz	3,000 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,600 MHz	2,600 MHz
AVX2	1,800 MHz	3,100 MHz	3,100 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,200 MHz	2,200 MHz
AVX512	1,200 MHz	2,900 MHz	2,900 MHz	2,500 MHz	2,500 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz

FIGURE 13 – Frequency Scaling on Xeon Gold 5120

Nous avons donc tout d'abord voulu comparer la vitesse d'exécution moyenne de chacune de nos versions dans un environnement multithreadé. Pour cela, nous monitorons /proc/cpuinfo pour chacun des coeurs de la machine.

	mean_freq	min_freq	max_freq	avg_min_freq	avg_max_freq	std_dev
avx512	2065.22	997.59	2700.01	1173.4	2322.19	225.04
avx2	2420.27	998.14	2700.49	1240.79	2622.83	270.16
opt	2435.12	998.55	2700.01	1099.93	2635.61	306.62
avx_balanced	2371.2	999.28	2700.09	1169.5	2680.59	316.25

FIGURE 14 – Statistiques de fréquence en fonction de la version

On constate bien un impact de l'utilisation d'AVX512 sur la fréquence d'exécution moyenne des coeurs.
Nous avons tout de même expérimenté avec une version "balanced" afin de voir l'impact du frequency scaling en général. Pour cela, on fait en sorte qu'un faible nombre de threads d'indice régulier lance une version avx512.

```
    int life_do_tile_avx_balanced (const int x, const int y, const int width,
                                   const int height)
{
  int tid = omp_get_thread_num ();
  if (tid % 12 == 0) {
    return life_do_tile_avx512 (x, y, width, height);
  } else {
    return life_do_tile_opt (x, y, width, height);
  }
}
```

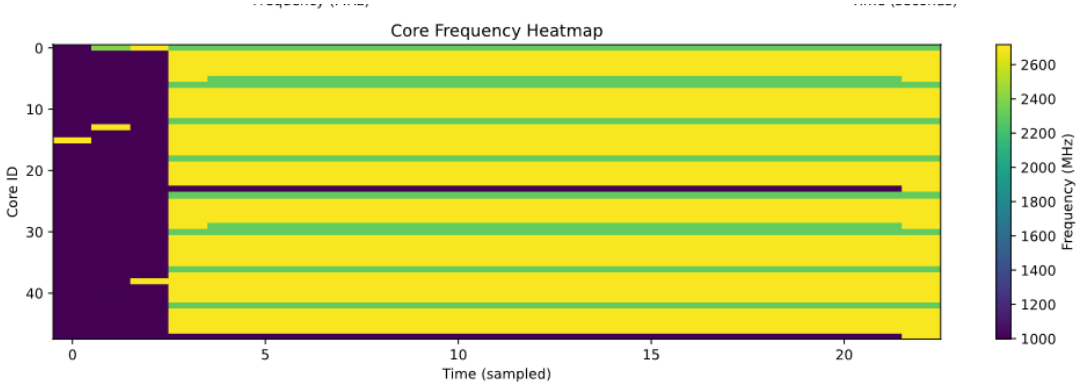

FIGURE 15 – Heatmap de la fréquence des coeurs

Même si la méthode peut être remise en question, nous constatons tout de même qu'un petit nombre de threads régulièrement disposé voit sa fréquence réduite, on peut sans trop prendre de risques en déduire que c'est l'impact du dynamic frequency scaling.

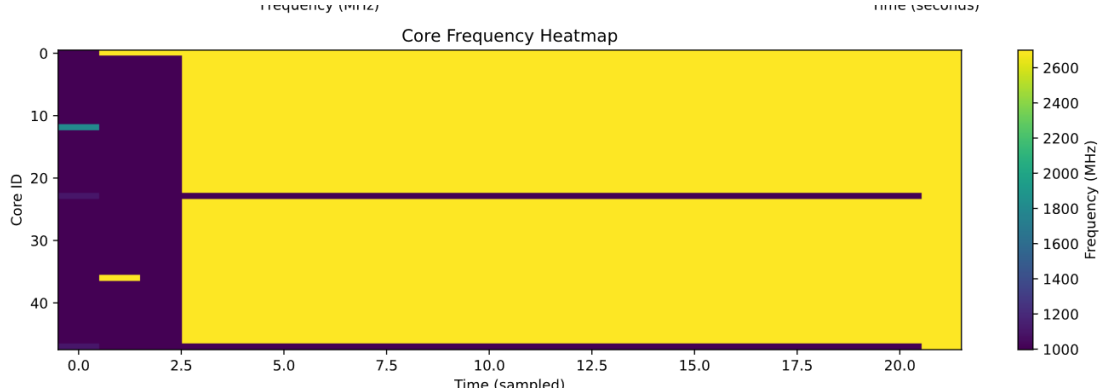

FIGURE 16 – Heatmap pour opt

La version opt quand-à-elle n'a pas ces baisses de fréquence.

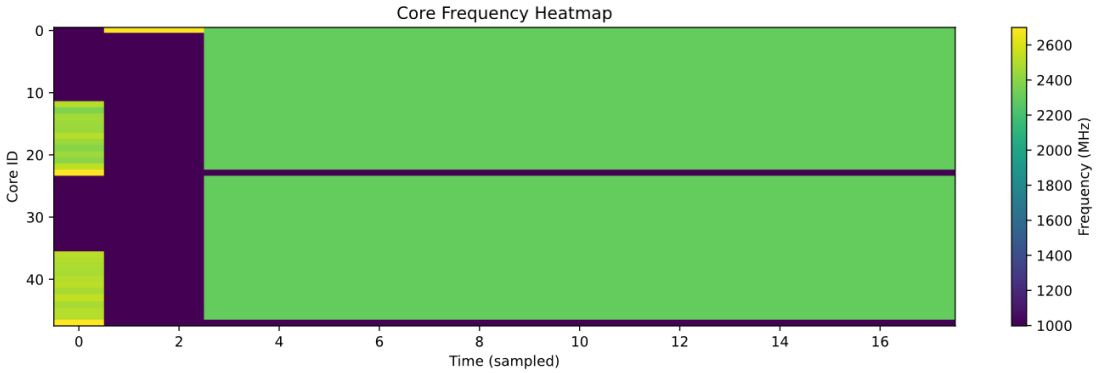

FIGURE 17 – Heatmap pour avx512

Alors que la version AVX512 elle est constamment abaissée.

3 Implémentation sur GPU avec OpenCL

Notons que pour plus de facilité, nous avons un noyau "life_gpu" pour toute la partie OpenCL.

3.1 Version de base

Nous avons d'abord implémenté une version simple du noyau OpenCL. Cette version applique les règles du jeu de la vie à la cellule associée au thread courant en effectuant directement les lectures et écritures dans la mémoire globale. Le jeu de la vie se prête d'ailleurs plutôt bien au passage sur GPU car non seulement le passage de la version CPU à la version GPU se fait sans vraiment réfléchir, mais également les accès mémoire lors du calcul de voisins semblent à première vu contigüs dans un warp.

```
__kernel void life_gpu_ocl (__global cell_t *in, __global cell_t *out)
{
  const unsigned x = get_global_id (0);
  const unsigned y = get_global_id (1);

  if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1) {
    const cell_t me = in[y * DIM + x];

    const unsigned n = in[(y - 1) * DIM + (x - 1)] + in[(y - 1) * DIM + x] +
                       in[(y - 1) * DIM + (x + 1)] + in[(y * DIM + (x - 1))] +
                       in[(y * DIM + (x + 1))] + in[(y + 1) * DIM + (x - 1)] +
                       in[(y + 1) * DIM + x] + in[(y + 1) * DIM + (x + 1)];
    const cell_t new_me = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
    out[y * DIM + x]    = new_me;
  }
}
```

Essayons dans un premier temps de trouver une taille de tuile optimale pour des problèmes en taille 1024 et 4096. Nous utiliserons la version avx2 en séquentiel pour calculer les speedups.

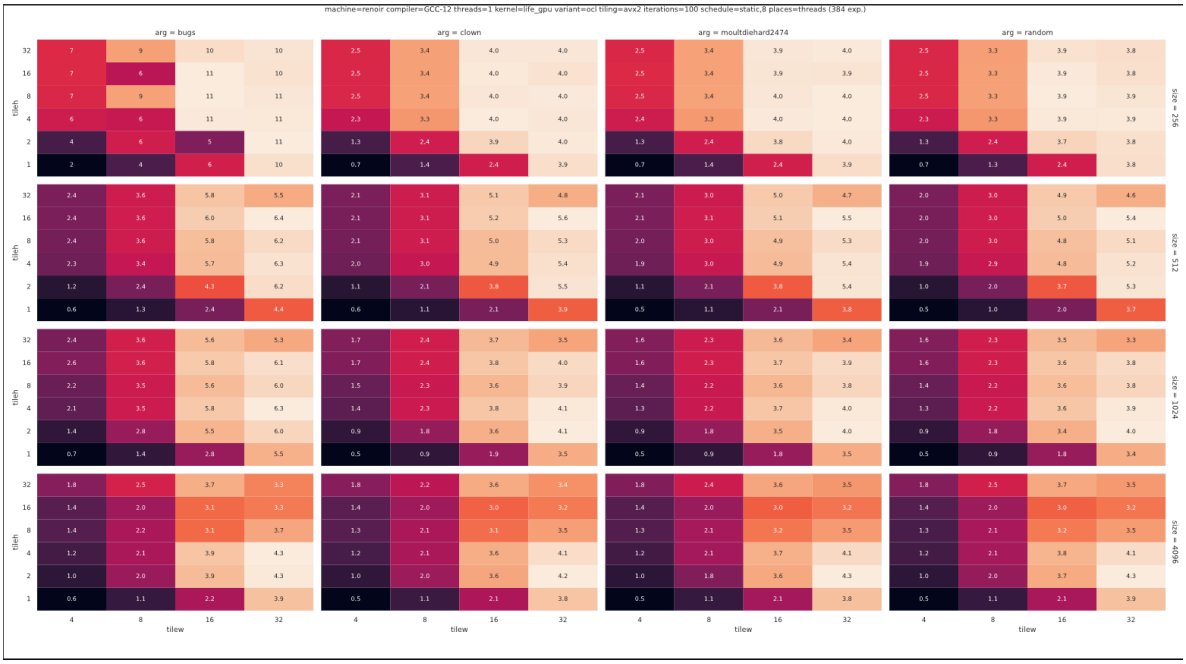

FIGURE 18 – Comparaison des performances : Version OpenCL de base vs meilleure version CPU

On constate des speedups face à la version AVX2 en séquentiel, qui nous laisse cependant penser qu'avec suffisament de coeurs notre version lazy_ompfor en AVX2 restera loin devant, mais surtout que le spot de tailles de tuiles les plus optimales est nettement moins étalé que sur CPU même si nous testons moins de valeur (notamment parce qu'on ne peut dépasser 1024 threads, la taille maximale d'un thread block). On peut probablement relier ce plus faible étalement au fait que notre version CPU apprécie surtout travailler sur de longues lignes, ce qui ne semble pas être le cas pour les versions GPU (nous avons effectués quelques tests type -tw=512, -th=2 et vice-versa et c'était toujours plus lent, voir beaucoup plus si tth est largement supérieur à -tw).

3.2 Quelques tentatives infructueuses

Un résumé des speedups (ou de leur absence) des versions que nous allons aborder sera présenté plus bas. Nous n'avons pas cherchés à optimiser le tuilage pour ces dernières car cette étape est assez longue sur GPU et nous n'attendons pas grand chose de ces versions dans tous les cas !

3.2.1 Version Vectorisation

Dans cette version, nous avons tentés d'expliciter la compilation en utilisant directement des types vectoriels (https ://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/vectorDataTypes.html), rendant la vectorisation encore plus explicite afin d'observer le comportement du GPU sur une telle exécution. Cependant, cette implémentation n'a pas porté ses fruits. Nous n'avons donc constaté aucune différence significative, ce qui est en soit plutôt cohérent étant donné que les GPU sont déjà des unités de calcul vectoriel, mais comme j'ai pu lire que cela pouvait parfois impacter les performances j'ai souhaité vérifier cela.

```
__kernel void life_gpu_ocl_more_explicit_vec (__global cell_t *in,
                                              __global cell_t *out)
{
  const unsigned x = get_global_id (0);
  const unsigned y = get_global_id (1);

  if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1) {
    const cell_t me = in[y * DIM + x];

    const uint3 line_above =
        (uint3)(in[(y - 1) * DIM + (x - 1)], in[(y - 1) * DIM + x],
                in[(y - 1) * DIM + (x + 1)]);
    const uint2 line_cell =
        (uint2)(in[(y * DIM + (x - 1))], in[(y * DIM + (x + 1))]);
```

```
    const uint3 line_below =
        (uint3)(in[(y + 1) * DIM + (x - 1)], in[(y + 1) * DIM + x],
                in[(y + 1) * DIM + (x + 1)]);

    uint3 identity_3 = (uint3)(1, 1, 1);
    uint2 identity_2 = (uint2)(1, 1);
    const unsigned n = line_above.x + line_above.y + line_above.z +
                       line_cell.x + line_cell.y + line_below.x + line_below.y +
                       line_below.z;
    const cell_t new_me = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
    out[y * DIM + x]    = new_me;
  }
}
```

3.2.2 Version Mémoire Locale

Nous avons implémenté une version qui utilise la mémoire locale où nous stockons la valeur de chaque pixel de chaque tuile et maintenons un halo afin d'éviter la divergence dans la gestion des bords. On constate un temps d'exécution plus long pour cette version. Notre kernel introduit également bien plus de divergence qu'il n'en évite, et vu la simplicité du calcul de l'état suivant dans le jeu de la vie, tout cet overhead n'est pas rentabilisé.

```
    __kernel void life_ocl (__global unsigned *in, __global unsigned *out)
{
  __local unsigned tile[TILE_H + 2][TILE_W + 2];

  unsigned x            = get_global_id (0);
  unsigned y            = get_global_id (1);
  unsigned xloc         = get_local_id (0) + 1;
  unsigned yloc         = get_local_id (1) + 1;
  unsigned local_size_x = get_local_size (0);
  unsigned local_size_y = get_local_size (1);

  unsigned width  = DIM;
  unsigned height = DIM;

  barrier (CLK_LOCAL_MEM_FENCE);

  if (x < width && y < height) {
    tile[yloc][xloc] = in[y * width + x];
  }

  if (xloc == 1) {
    unsigned left_x = (x == 0) ? width - 1 : x - 1;
    tile[yloc][0]   = in[y * width + left_x];
  }

  if (xloc == local_size_x) {
    unsigned right_x              = (x == width - 1) ? 0 : x + 1;
    tile[yloc][local_size_x + 1] = in[y * width + right_x];
  }

  if (yloc == 1) {
    unsigned top_y = (y == 0) ? height - 1 : y - 1;
    tile[0][xloc]  = in[top_y * width + x];
  }

  if (yloc == local_size_y) {
    unsigned bottom_y             = (y == height - 1) ? 0 : y + 1;
    tile[local_size_y + 1][xloc] = in[bottom_y * width + x];
  }

  if (xloc == 1 && yloc == 1) {
    unsigned tl_x = (x == 0) ? width - 1 : x - 1;
    unsigned tl_y = (y == 0) ? height - 1 : y - 1;
    tile[0][0]    = in[tl_y * width + tl_x];
  }
  if (xloc == local_size_x && yloc == 1) {
    unsigned tr_x             = (x == width - 1) ? 0 : x + 1;
    unsigned tr_y             = (y == 0) ? height - 1 : y - 1;
    tile[0][local_size_x + 1] = in[tr_y * width + tr_x];
  }
  if (xloc == 1 && yloc == local_size_y) {
    unsigned bl_x             = (x == 0) ? width - 1 : x - 1;
    unsigned bl_y             = (y == height - 1) ? 0 : y + 1;
    tile[local_size_y + 1][0] = in[bl_y * width + bl_x];
  }
  if (xloc == local_size_x && yloc == local_size_y) {
    unsigned br_x                             = (x == width - 1) ? 0 : x + 1;
    unsigned br_y                             = (y == height - 1) ? 0 : y + 1;
    tile[local_size_y + 1][local_size_x + 1] = in[br_y * width + br_x];
  }
```

```
61
   barrier (CLK_LOCAL_MEM_FENCE);
63
   if (x < width && y < height) {
65   unsigned neighbors = tile[yloc - 1][xloc - 1] + tile[yloc - 1][xloc] +
                          tile[yloc - 1][xloc + 1] + tile[yloc][xloc - 1] +
67                        tile[yloc][xloc + 1] + tile[yloc + 1][xloc - 1] +
                          tile[yloc + 1][xloc] + tile[yloc + 1][xloc + 1];
69
     unsigned current   = tile[yloc][xloc];
71   unsigned new_state = 0;

73   if (current == 1) {
       new_state = (neighbors == 2 || neighbors == 3) ? 1 : 0;
75   } else {
       new_state = (neighbors == 3) ? 1 : 0;
77   }

79   out[y * width + x] = new_state;
   }
81 }
```

3.2.3 Parce qu'on ne sait jamais...

Nous avons également testé une optimisation que nous avions essayée sur CPU et qui n'avait eu aucun impact, pour voir si cela serait différent sur GPU. Étant donné que DIM est toujours une puissance de 2, nous avons pensé que nous pourrions utiliser des décalages à gauche pour les multiplications. Nous avons donc développé un kernel "binmul" qui prend en 3ème paramètre la puissance de 2 correspondant à DIM, mais là encore, aucun gain significatif n'a été observé.

```
1  __kernel void life_gpu_ocl_binmul (__global cell_t *in, __global cell_t *out,
                                      const unsigned shift_by)
3  {
     const unsigned x = get_global_id (0);
5    const unsigned y = get_global_id (1);

7    if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1) {
       const cell_t me = in[(y << shift_by) + x];
9
       const unsigned n =
11         in[((y - 1) << shift_by) + (x - 1)] + in[((y - 1) << shift_by) + x] +
           in[((y - 1) << shift_by) + (x + 1)] + in[((y << shift_by) + (x - 1))] +
13         in[((y << shift_by) + (x + 1))] + in[((y + 1) << shift_by) + (x - 1)] +
           in[((y + 1) << shift_by) + x] + in[((y + 1) << shift_by) + (x + 1)];
15     const cell_t new_me      = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
       out[(y << shift_by) + x] = new_me;
17   }
   }
```

3.2.4 Comparaison des performances

FIGURE 19 – Comparatif de performances sur nos différentes expérimentations

Nous constatons cela au moment où nous écrivons le rapport. Jusqu'à maintenant nous pensions que nos petites expériences performaient toujours largement au deça de la version ocl naïve, mais il semblerait qu'en utilisant cette version naïve comme référence, nous puissions avoir de très (très) légers speedups. La version "binmul" semble systématiquement légèrement plus rapide, la version vectorisée semble pouvoir être légèrement plus rapide (même s'il pourrait s'agir d'un bruit dans nos mesures)... C'est surprenant !

3.3 2 cellules en 1

Nous avons ensuite tenté de traiter deux cellules au lieu d'une seule. Nous avons divisé le nombre de threads par 2 puis traité la mémoire de façon à toujours avoir un accès contigu, comme l'exemple donné dans le cours. Nous avons longtemps pensés qu'il s'agissait de la seule optimisation un tant soit peu efficace, mais finalement il semblerait qu'elle soit légèrement moins efficace que binmul. Nous pourrions envisager un noyau binmul + 2x, et nous aurions probablement de bon résultats sur une configuration random, mais nous nous contenterons de la version lazy.

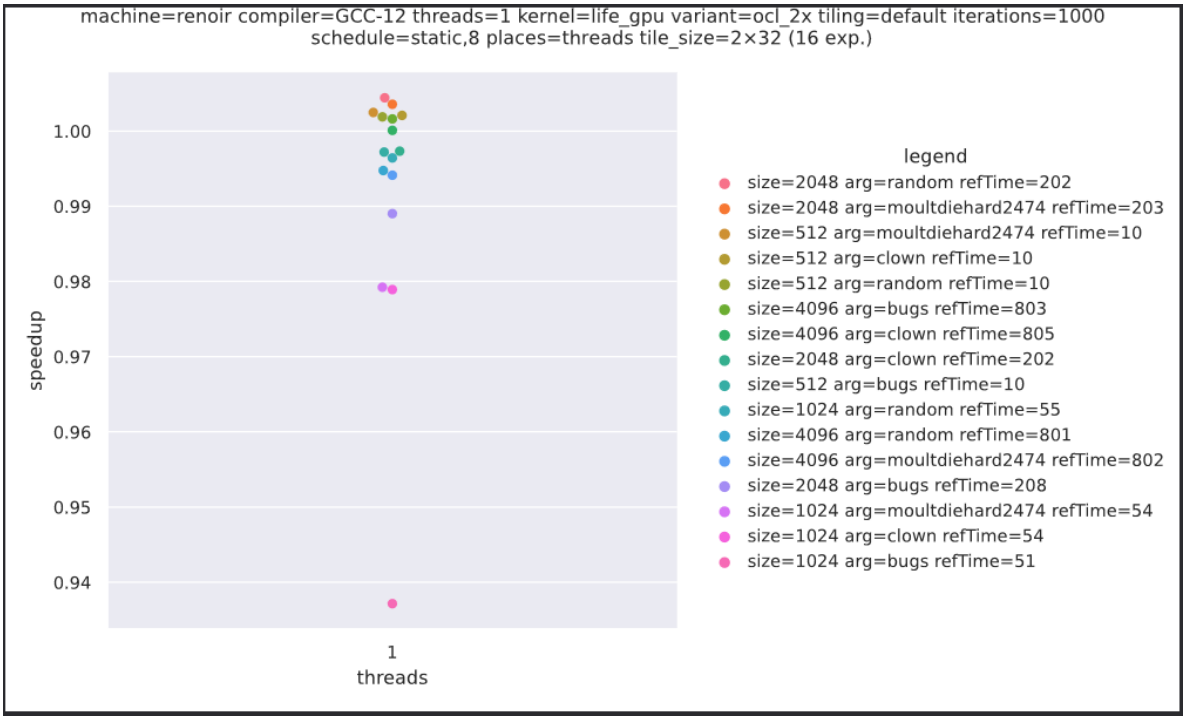

FIGURE 20 – Performances du traitement de deux cellules par thread vs une cellule

3.4 Version Paresseuse sur GPU

Enfin, la version Lazy nous a donné du fil à retordre. Nous nous sommes inspirés de la version que nous avons produite sur CPU, avec deux tableaux de tuiles : on indique à nos voisins de changer quand on change. Notre problème venait d'une mauvaise compréhension du paradigme de programmation GPU. En effet, nous remettions à zéro le tableau "tile_out" au début du kernel, et tout fonctionnait jusqu'à une certaine taille d'image. Nous en avons déduit qu'il s'agissait d'un problème de concurrence, où des threads exécutant le début du kernel écrasaient les valeurs placées par des threads exécutés à la fin du kernel. La première solution a été de faire deux kernels, un premier pour mettre tile_out à zéro et un deuxième pour le calcul, mais cela s'est montré peu efficace car cela ajoutait beaucoup d'overhead. Par la suite, nous avons compris que notre version lazy n'était pas optimale et que nous pouvions effectuer ce reset à la fin du calcul des tuiles calculées et ainsi éviter tout problème de concurrence. C'est ce que nous avons fait et cela s'est avéré très efficace.

```
__kernel void life_gpu_ocl_lazy (__global cell_t *in, __global cell_t *out,
                                 __global cell_t *tile_in,
                                 __global cell_t *tile_out)
{
  unsigned x          = get_global_id (0);
  unsigned y          = get_global_id (1);
  unsigned xloc       = get_local_id (0);
  unsigned yloc       = get_local_id (1);
  unsigned xgroup     = get_group_id (0);
  unsigned ygroup     = get_group_id (1);
  unsigned NB_TILES_W = DIM / TILE_W;
  unsigned xtile      = xgroup + 1;
  unsigned ytile      = ygroup + 1;
  unsigned tile_idx   = ytile * NB_TILES_W + xtile;
  __local unsigned compute_tile;
  __local unsigned tile_change;
  // first of the warp (so first of the tile as well)
  if (xloc == 0 && yloc == 0) {
    tile_change  = 0;
    compute_tile = tile_in[tile_idx];
  }
  barrier (CLK_LOCAL_MEM_FENCE);
  if (!compute_tile)
    return;
  if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1) {
    const cell_t me  = in[y * DIM + x];
    const unsigned n = in[(y - 1) * DIM + (x - 1)] + in[(y - 1) * DIM + x] +
                       in[(y - 1) * DIM + (x + 1)] + in[(y * DIM + (x - 1))] +
```

```
                        in[(y * DIM + (x + 1))] + in[(y + 1) * DIM + (x - 1)] +
                        in[(y + 1) * DIM + x] + in[(y + 1) * DIM + (x + 1)];
      const cell_t new_me = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
      if (new_me != me) {
        tile_change = true;
      }
      out[y * DIM + x] = new_me;
    }
    barrier (CLK_LOCAL_MEM_FENCE);
    if (yloc == 0 && xloc == 0) {
      if (tile_change) {
        tile_out[tile_idx]                                  = 1;
        tile_out[ytile * NB_TILES_W + (xtile + 1)]          = 1;
        tile_out[(ytile - 1) * NB_TILES_W + (xtile - 1)] = 1;
        tile_out[(ytile - 1) * NB_TILES_W + xtile]          = 1;
        tile_out[(ytile - 1) * NB_TILES_W + (xtile + 1)] = 1;
        tile_out[ytile * NB_TILES_W + (xtile - 1)]          = 1;
        tile_out[(ytile + 1) * NB_TILES_W + (xtile - 1)] = 1;
        tile_out[(ytile + 1) * NB_TILES_W + xtile]          = 1;
        tile_out[(ytile + 1) * NB_TILES_W + (xtile + 1)] = 1;
      }
      tile_in[tile_idx] = 0;
    }
}
```

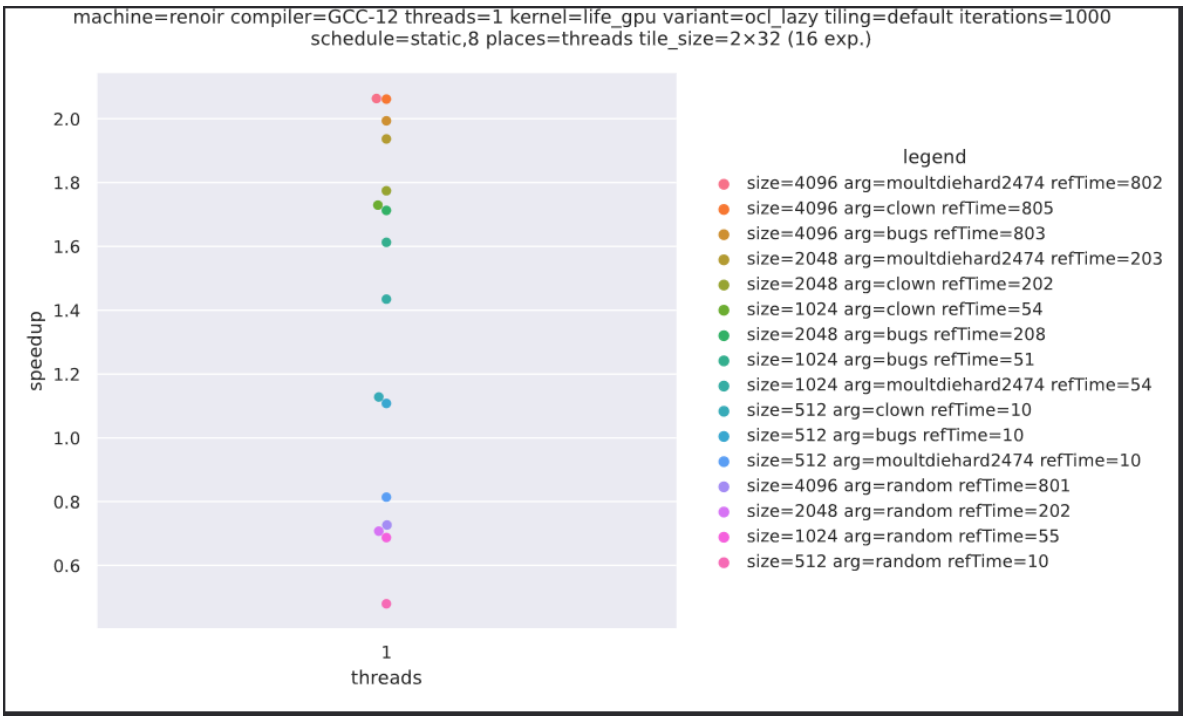

FIGURE 21 – Performance de l'évaluation paresseuse sur GPU vs version standard

On constate donc un sérieux gain de performances sur les configurations lazy-friendly, jusqu'à un facteur 2 et pas seulement pour clown. Les speedups sont peut-être un peu moins impressionnants que sur la version CPU, mais différents facteurs peuvent probablement expliquer cela, notamment l'impact de la divergence induit par le non calcul d'une tuile, ou encore l'impact de la synchronisation au niveau des groupes (là où notre version CPU ne requiérait pas de synchronisation).

4 Comparaison GPU/AVX

Analysons maintenant quelques runs afin de comparer la rapidité d'exécution entre les versions lazy_ompfor + AVX2 et OpenCL.

Commençons par une taille plutôt faible :

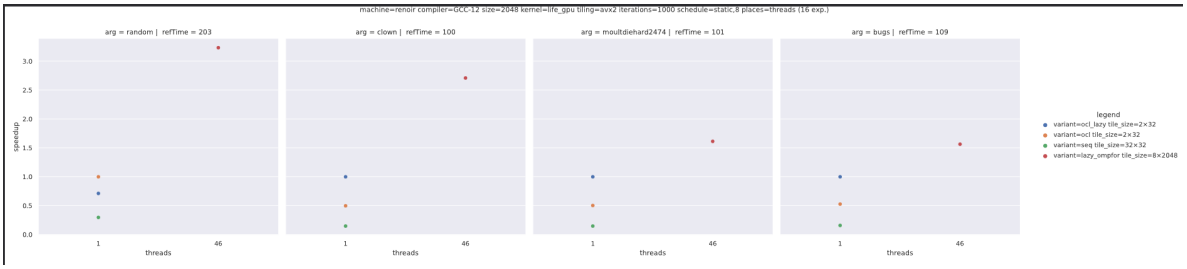

FIGURE 24 – Comparaison Lazy OMP vs Lazy OCL en taille 2048

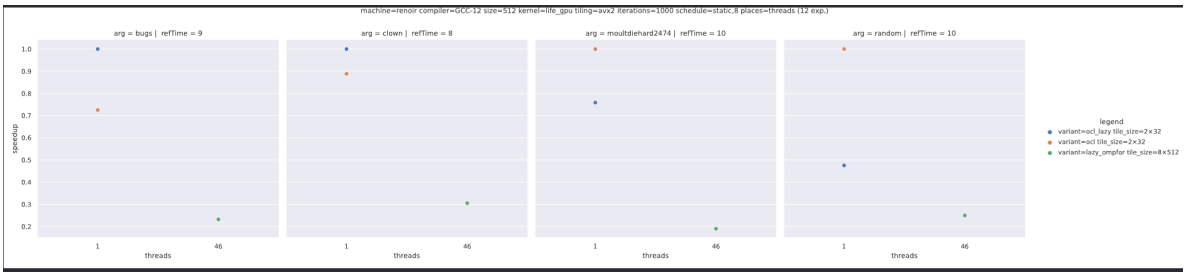

FIGURE 22 – Comparaison Lazy OMP vs Lazy OCL en taille 512

5 Conclusion

On constate un net avantage pour les versions ocl. La version Lazy l'emporte quand la configuration lui en laisse l'occasion, sinon c'est la version naïve (seule autre version GPU testée ici). On peut comprendre pourquoi elle l'emporte sur la version random, qui a une forte entropie et est donc peu efficace dans les cas d'évaluation paresseuse. Pour la run moultdiehard2474, on peut supposer que le tuilage n'était pas très adapté à l'évaluation paresseuse étant donné que nous n'avons pas prit la peine de fine-tuner les paramètres.

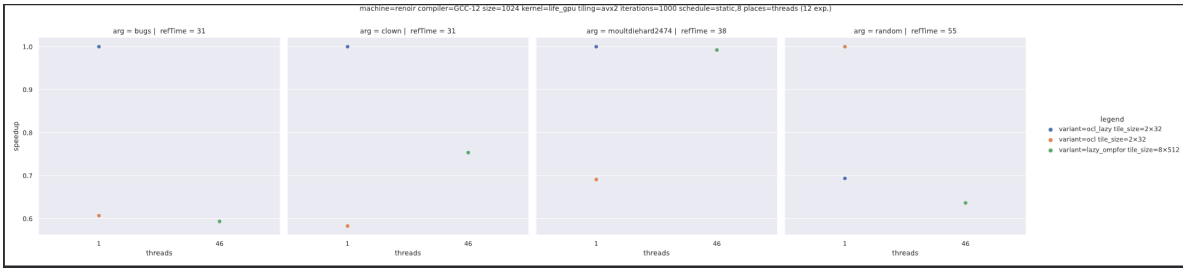

FIGURE 23 – Comparaison Lazy OMP vs Lazy OCL en taille 1024

En taille 1024, l'avantage est déjà nettement moin marqué pour les versions OpenCL, sauf pour la run "bugs" avec la version paresseuse qui est toujours loin devant. Sinon, la version OCL naïve n'a plus la même avance sur les version lazy. Il aurait pu être intéressant de mettre en plus une version non paresseuse omp.* En taille 2048 et plus en revanche la version lazy_ompfor devance toutes les runs OpenCL. Plusieurs hypothèses peuvent donner un début d'explication à cela. Déjà, pour une run sur une version OpenCL on alloue un thread par pixel de la tuile. Si l'on est sur une tuile que l'on ne calcul pas, tous ces threads seront préparés (bien que cela soit peu couteux) pour au final ne rien calculer. En CPU, un thread choisit si l'entièreté d'une cuile va être calculée ou non.
Egalement, les caches L2 et L3 (absent sur GPU sauf erreur de notre part pour le L3) suffisent à mettre des lignes (voir plusieurs lignes) à l'intérieur ce qui n'est pas le cas sur GPU. Nous serions curieux cependant d'avoir lors de la correction plus d'éléments pouvant expliquer ces comportements. Nous n'incluons pas les graphs de plus grosses configurations car le comportement est le même (mais l'écart

est réduit).

5.1 Bilan des Optimisations

5.1.1 Vectorisation CPU

La vectorisation par instructions SIMD s'est révélée efficace mais avec des nuances importantes :
— L'approche AVX2 a démontré un gain de performance significatif sur les grandes images.
— Le passage à AVX512 n'a pas toujours offert l'amélioration attendue, peut-être en raison du frequency scaling qui limite la fréquence des cœurs mais il y a très probablement d'autres raisons.
— Notre tentative d'approche "balanced" (distribution des instructions AVX2/AVX512 entre les threads) n'a pas permis de contourner efficacement cette limitation.
— La gestion des bords reste un défi pour la vectorisation.

5.1.2 Implémentation GPU

Le portage sur GPU a révélé plusieurs aspects intéressants :
— La version de base offre déjà des gains substantiels pour les petites à moyennes grilles, démontrant l'adéquation naturelle entre le parallélisme du jeu et l'architecture GPU.
— Contrairement au CPU, l'approche vectorisée explicite n'apporte pas vraiment d'avantages, le GPU étant déjà optimisé pour les traitements de grandes images et a en soit déjà une approche vectorielle du traitement des données.
— L'utilisation de la mémoire locale avec les halos s'est avérée contre-productive, l'overhead d'initialisation et de synchronisation dépassant les bénéfices pour un calcul aussi simple. Cependant nous avons peut-être mal utilisé la mémoire locale, où alors l'ordonnancement des threads + les accès contigus ont masqués cette latence.
— Le traitement de deux cellules par thread s'est révélé être relativement peu efficace, même si un léger speedup a été remarqué.
— La version paresseuse a nécessité une adaptation au modèle d'exécution GPU, et nous a permi de trouver une amélioration possible dans notre version CPU.

5.2 Perspectives

Plusieurs pistes prometteuses pourraient être explorées dans le futur :
— Développement d'une solution hybride CPU-GPU qui répartirait la charge en sur la grille (etape-4), en calculant le nombre optimal de pixels pouvant être calculés sur carte graphique et sur GPU.
— Implémentation d'une approche de masquage efficace pour la vectorisation des bords.
— Adaptation dynamique de la stratégie d'exécution en fonction des caractéristiques de la machine cible (présence d'AVX512, puissance du GPU, etc.)
— Investigation de formats de données plus compacts, potentiellement avec des représentations bit-à-bit / bitset pour réduire davantage l'empreinte mémoire, et peut-être même bénéficier d'algorithmes type SWAR etc.

En définitive, ce projet illustre comment l'optimisation des performances nécessite une approche en tatônement tout en appliquant les notions de cours et des recherches personnels.

6 Bonus

Un peu déçus des speedups obtenus avec le kernel utilisant la mémoire locale, nous pouvons au moins nous consoler avec ces deux variantes de life3d (disponibles sur la branche éponyme) faites après le TD sur heat3d pour constater de réels speedups ! En espérant voir un jour d'autres mesh que le torus (car en trouver est mission impossible et en faire m'a rappelé à quel point blender est peu avenant...)

```
    __kernel void life3d_ocl_naive (__global float *in, __global float *out,
                                    __global int *neighbors,
                                    __global int *index_neighbor)
{
  const int index = get_global_id (0);

  if (index < NB_CELLS) {
    int cur = index;

    int me = in[cur];

    int top       = neighbors[index_neighbor[cur]];
    int top_right = neighbors[index_neighbor[top] + 1];
    int top_left  = neighbors[index_neighbor[top] + 3];

    int right = neighbors[index_neighbor[cur] + 1];
    int left  = neighbors[index_neighbor[cur] + 3];

    int bottom       = neighbors[index_neighbor[cur] + 2];
    int bottom_right = neighbors[index_neighbor[bottom] + 1];
    int bottom_left  = neighbors[index_neighbor[bottom] + 3];

    int n = in[top] + in[top_right] + in[top_left] + in[right] + in[left] +
            in[bottom] + in[bottom_right] + in[bottom_left];

    const char new_me = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
    out[cur]          = new_me;
  }
}

__kernel void life3d_ocl (__global float *in, __global float *out,
                          __global int *neighbor_soa)
{
  const int index = get_global_id (0);
  out[index]      = 0;
  if (index < NB_CELLS) {

    int me = in[index];

    int top       = neighbor_soa[index];
    int top_right = neighbor_soa[1 * NB_CELLS + top];
    int top_left  = neighbor_soa[3 * NB_CELLS + top];

    int right = neighbor_soa[1 * NB_CELLS + index];
    int left  = neighbor_soa[3 * NB_CELLS + index];

    int bottom       = neighbor_soa[2 * NB_CELLS + index];
    int bottom_right = neighbor_soa[1 * NB_CELLS + bottom];
    int bottom_left  = neighbor_soa[3 * NB_CELLS + bottom];

    int n = in[top] + in[top_right] + in[top_left] + in[right] + in[left] +
            in[bottom] + in[bottom_right] + in[bottom_left];

    out[index] = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
  }
}
static inline bool __in (int a, int b, int c)
{
  return c >= a && c <= b;
}
__kernel void life3d_ocl_cache (__global float *in, __global float *out,
                                __global int *neighbor_soa)
{
  const int index       = get_global_id (0);
  const int group_index = get_group_id (0);
  const int group_start = group_index * TILE;
  const int group_end   = group_start + TILE - 1;
  const int loc_index   = get_local_id (0);
  __local float tile[TILE];
  if (index < NB_CELLS) {
    tile[loc_index] = in[index];
    barrier (CLK_LOCAL_MEM_FENCE);
    int me = in[index];

    int top       = neighbor_soa[index];
    int top_right = neighbor_soa[1 * NB_CELLS + top];
    int top_left  = neighbor_soa[3 * NB_CELLS + top];

```

```
    int right = neighbor_soa[1 * NB_CELLS + index];
    int left  = neighbor_soa[3 * NB_CELLS + index];

    int bottom       = neighbor_soa[2 * NB_CELLS + index];
    int bottom_right = neighbor_soa[1 * NB_CELLS + bottom];
    int bottom_left  = neighbor_soa[3 * NB_CELLS + bottom];
    int n;
    if (__in (group_start, group_end, top_left) &&
        __in (group_start, group_end, top_right) &&
        __in (group_start, group_end, bottom_left) &&
        __in (group_start, group_end, bottom_right)) {
      n = tile[top - group_start] + tile[top_right - group_start] +
          tile[top_left - group_start] + tile[right - group_start] +
          tile[left - group_start] + tile[bottom - group_start] +
          tile[bottom_right - group_start] + tile[bottom_left - group_start];

    } else {
      n = in[top] + in[top_right] + in[top_left] + in[right] + in[left] +
          in[bottom] + in[bottom_right] + in[bottom_left];
    }

    out[index] = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
  }
}
```