Programmation Paralléle

Lucas Marques, Matis Duval

March 17, 2025

Abstract

Rapport Projet: Le jeu de la vie - Deuxième Jalon

1 Introduction

Suite au premier jalon où nous avons explorés les stratégies de parallélisation de base, ce deuxième jalon se concentre sur l'optimisation des calculs internes aux tuiles et l'implémentation d'une évaluation paresseuse, ainsi que la prise en compte des différentes architectures de processeurs disponibles au CREMI (NUMA avec plusieurs noeuds en 008 et architecture hétérogène en 009). Notre objectif est d'améliorer davantage les performances de notre simulation du Jeu de la Vie en exploitant les caractéristiques architecturales des machines cibles ainsi que les caractéristiques spatiales de la simulation (présence de zones mortes).

2 Version Optimisé do_tile

L'optimisation des calculs internes aux tuiles vise à maximiser l'efficacité du pipeline en réduisant les branchements conditionnels et les sauts, notamment en déroulant les boucles pour permettre de meilleures optimisations par le compilateur. Nous avons d'abord déroulés la boucle de calcul de nombre de voisins, puis appliqués des techniques de branchless programming afin de se défaire au maximum des sauts conditionnels.

2.1 Implémentation de life_do_tile_opt()

Voici notre version optimisée du calcul interne aux tuiles :

```
int life_do_tile_opt (const int x, const int y, const int width, const int height){
  char change  = 0;
  // precomputing start and end indexes of tile's both width and height
  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;

  for (int i = y_start; i < y_end; i++) {
    for (int j = x_start; j < x_end; j++) {
      const char me = cur_table (i, j);

      // we unrolled the loop and check it in lines
      const char n = cur_table(i-1, j-1) + cur_table(i-1, j) + cur_table(i-1, j+1)
        + cur_table(i, j-1) + cur_table(i, j+1) + cur_table(i+1, j-1)
        + cur_table(i+1, j) + cur_table(i+1, j+1);
      // while we are at it, we apply some simple branchless programming logic
      const char new_me = (me & ((n == 2) | (n == 3))) | (!me & (n == 3));
      change |= (me ^ new_me);
      next_table (i, j) = new_me;
    }
  }
  return change;
}
```

Les principales optimisations implémentées sont :

- Déroulement de boucles pour traiter plusieurs cellules à la fois
- Élimination des branchements conditionnels par l'usage d'opérations arithmétiques
- Accès mémoire optimisés pour améliorer la localité spatiale (en travaillant sur des lignes)
- Pré-calcul des conditions de bord pour éviter de répéter les vérifications et/ou les calculs dans les boucles.

  De plus, en utilisant -fopt-info-vec nous constatons que GCC vectorise au niveau de notre boucle de code, ce qui semblait se confirmer en observant l'output assembleur sur godbolt.org.

2.2 Analyse des Performances

```
kernel/c/life.c:153:29: optimized: loop vectorized using 32 byte vectors
kernel/c/life.c:153:29: optimized: loop vectorized using 16 byte vectors
```

Figure 1: optimisations par le compileur

Nous observons un gain de performance significatif avec notre fonction optimisée. Sur la machine de la salle 008, nous atteignons une acceleration en séquentiel par rapport à la version par défaut. Ce gain peut s'expliquer par :

- Une meilleure utilisation du cache due à l'accès contigu aux données.
- La réduction significative des instructions conditionnelles permettant un meilleur fonctionnement du prédicteur de branchement, ainsi que l'évitement d'un certain nombre de sauts.

Par la suite, il sera envisageable d'implémenter la vectorisation des opérations dans nos do_tile.

Nous constatons une forte différence, d'un facteur d'environ 4, entre la version optimisée et la version par défaut.

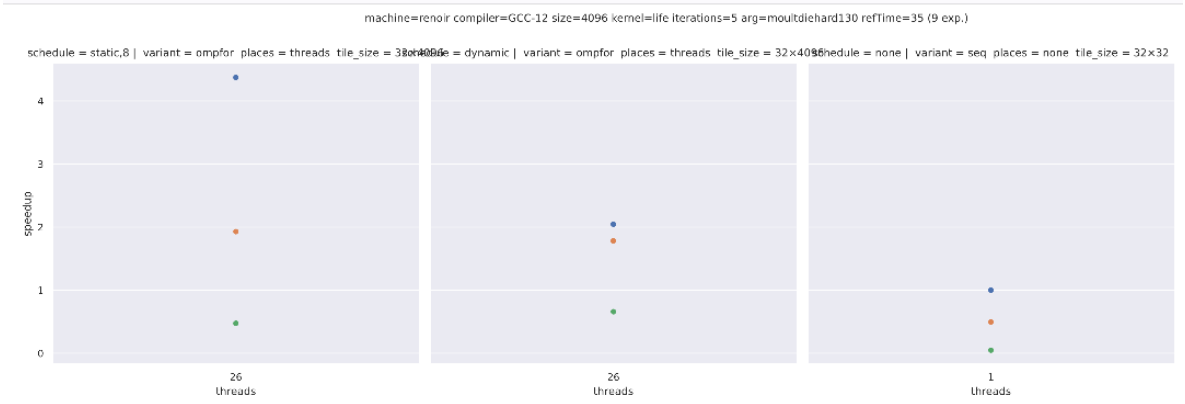


Figure 2: comparaison default vs opt

Et c'est tout aussi évident sur cette trace, si l'on fait abstraction des latences d'accès à la mémoire, on voit bien que l'on calcule les tuiles beaucoup plus rapidement.

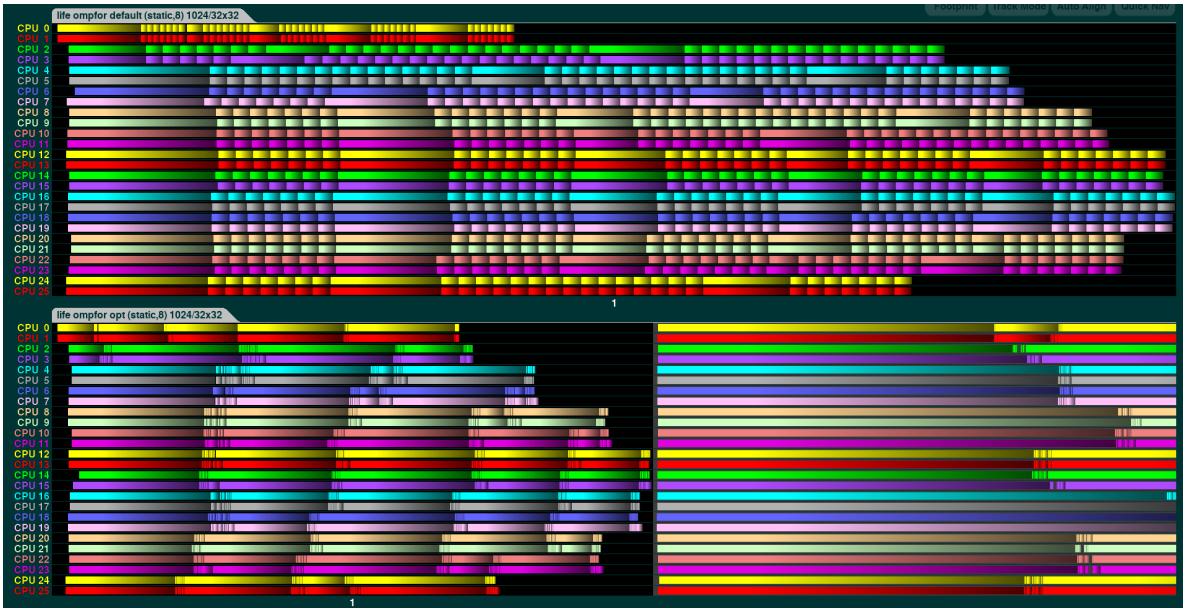


Figure 3: Au dessus la version par défaut, en dessous la version optimisée

3 Implémentation de l'Évaluation Paresseuse

L'évaluation paresseuse exploite le fait que certaines régions du domaine restent stables durant plusieurs itérations. Nous évitons ainsi de recalculer inutilement ces tuiles.

3.1 Version Séquentielle (life_compute_lazy)

```
unsigned life_compute_lazy(unsigned nb_iter) {
  unsigned res = 0;

  for (int it = 1; it <= nb_iter; it++) {
    unsigned change = 0;
    for (int y = 0; y < DIM; y += TILE_H) {
      unsigned tile_y = y / TILE_H;
      for (int x = 0; x < DIM; x += TILE_W) {
        unsigned local_change = 0;
        unsigned tile_y = y / TILE_H;
        unsigned tile_x = x / TILE_W;
        // checking if we should recompute this tile or not
        if (cur_dirty(tile_y, tile_x)) {
          // we need to keep track of per-tile changes
          local_change = do_tile(x, y, TILE_W, TILE_H);
          change |= local_change;

          if (local_change) {
            // setting them to 2 in order to avoid writing 0 on a unchanged tile that has some changes i
            next_dirty(tile_y-1, tile_x-1) = 2;
            next_dirty(tile_y-1, tile_x)   = 2;
            next_dirty(tile_y-1, tile_x+1) = 2;
            next_dirty(tile_y, tile_x-1)   = 2;
            next_dirty(tile_y, tile_x)     = 1;  // except for the one of the iteration
            next_dirty(tile_y, tile_x+1)   = 2;
            next_dirty(tile_y+1, tile_x-1) = 2;
            next_dirty(tile_y+1, tile_x)   = 2;
            next_dirty(tile_y+1, tile_x+1) = 2;
          } else {
            if (next_dirty(tile_y, tile_x) != 2) {  // checking if needs to be recomputed for a neighbor
              next_dirty(tile_y, tile_x) = 0;
              cur_dirty(tile_y, tile_x) = 0;
            }
          }
        }
      }
    }
```

```
38      if(!change) return it;
        swap_tables_w_dirty();
40    }

42    return res;
  }
```

Ici, nous maintenons deux tableaux. Un qui sera mit à jour par les tuiles elles-même et un qui sera mit à jour par les voisins. Cela nous permet d'éviter d'utiliser de la synchronisation car deux accès concurrents écriraient la même valeur.
De plus, nous utilisons un tableau ajoutant une ligne de chaque côté afin de nous autoriser des out-of-bound writes. Comme ça, pas besoin de tester si l'on est sur une bordure ou non. L'objectif affiché était clairement d'éviter au maximum tout saut conditionnel et toute synchronisation afin d'éviter tout surcoût. Nous constatons directement sur la heatmap l'impact de notre optimisation:

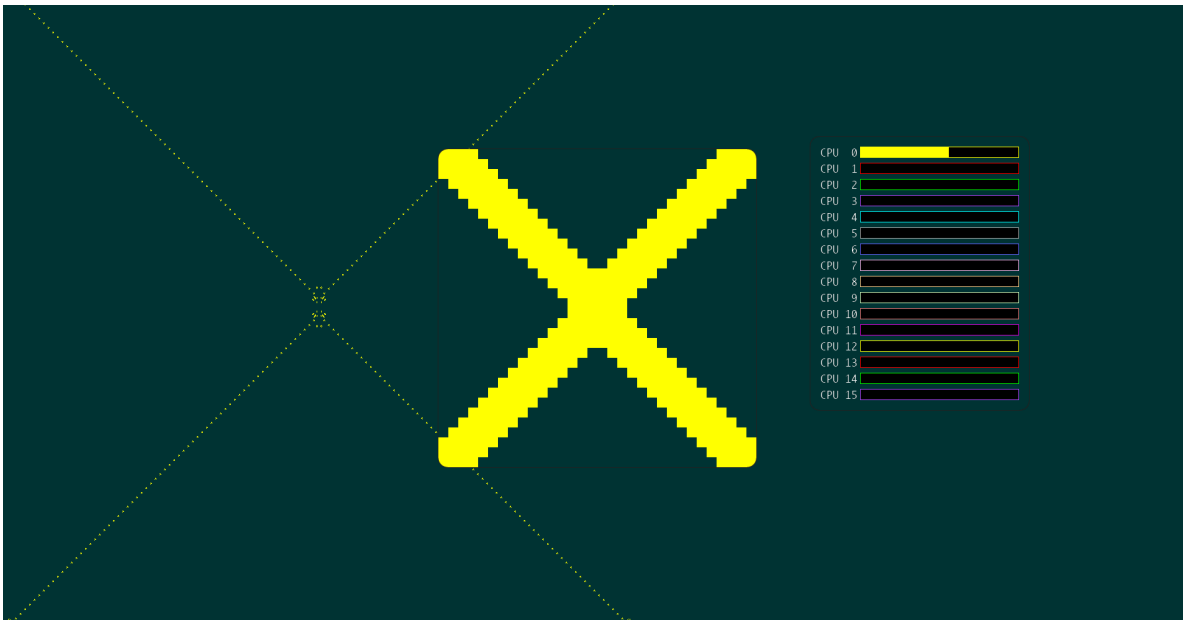


Figure 4: Heatmap de la simulation par défaut en taille 1024

3.2 Version Parallèle (life_compute_omp_lazy)

```
1  unsigned life_compute_lazy_ompfor(unsigned nb_iter) {
     unsigned res = 0;
3
     for (int it = 1; it <= nb_iter; it++) {
5      unsigned change = 0;
       #pragma omp parallel for reduction(|: change) collapse(2) schedule(runtime)
7      for (int y = 0; y < DIM; y += TILE_H) {
         for (int x = 0; x < DIM; x += TILE_W) {
9          unsigned local_change = 0;
           unsigned tile_y = y / TILE_H;
11         unsigned tile_x = x / TILE_W;
           // checking if we should recompute this tile or not
13         if (cur_dirty(tile_y, tile_x) || next_dirty(tile_y, tile_x)) {
             // we need to keep track of per-tile changes
15           local_change = do_tile(x, y, TILE_W, TILE_H);
             change |= local_change;
17
             if (local_change) {
19             next_dirty(tile_y-1, tile_x-1) = 2;
               next_dirty(tile_y-1, tile_x)   = 2;
21             next_dirty(tile_y-1, tile_x+1) = 2;
               next_dirty(tile_y, tile_x-1)   = 2;
23             next_dirty(tile_y, tile_x)     = 1;  // except for the one of the iteration
               next_dirty(tile_y, tile_x+1)   = 2;
25             next_dirty(tile_y+1, tile_x-1) = 2;
               next_dirty(tile_y+1, tile_x)   = 2;
27             next_dirty(tile_y+1, tile_x+1) = 2;
```

```
            } else {
29            if (next_dirty(tile_y, tile_x) != 2) {  // checking if needs to be recomputed for a neighbor
                cur_dirty(tile_y, tile_x) = 0;
31            }
            }
33        }
        }
35    }

37    if(!change) return it;
      swap_tables_w_dirty();
39  }

41  return res;
}
```

Rien de bien complexe, nous parallélisons simplement la boucle étant donné que le code est supposément concurrence-friendly. Il aurait pu être intéressant d'envisager une autre structure de donnée pour stocker l'état des tuiles à recalculer, un quadtree aurai par exemple permit de subdiviser l'espace en séparant efficacement les zones mortes des zones avec de l'activité tout en ayant peu d'impact mémoire.
En regardant les heatmaps lors de l'exécution, on constate bien l'impact de l'évaluation lazy répartie sur les différents coeurs:

4 Analyse des Performances Lazy

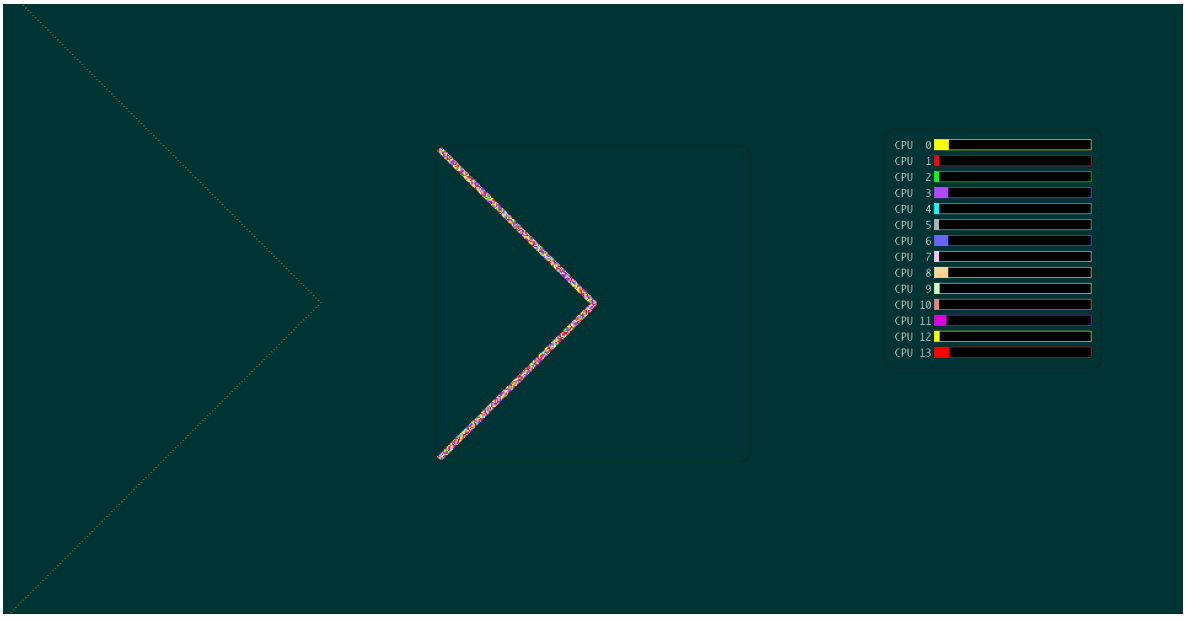


Figure 5: Recherche d'un tuilage optimal

Nous pouvons voir ici que nous avons un tilling qui est plutôt optimal en travaillant avec des lignes, de plus comme la version lazy traite une plus grande multitude de cas alors elle est plus optimale. Cela nous étonne. En effet, nous pensions que des tuiles "cache-friendly" seraient meilleurs. Nous reviendrons sur cela à la fin du rapport.

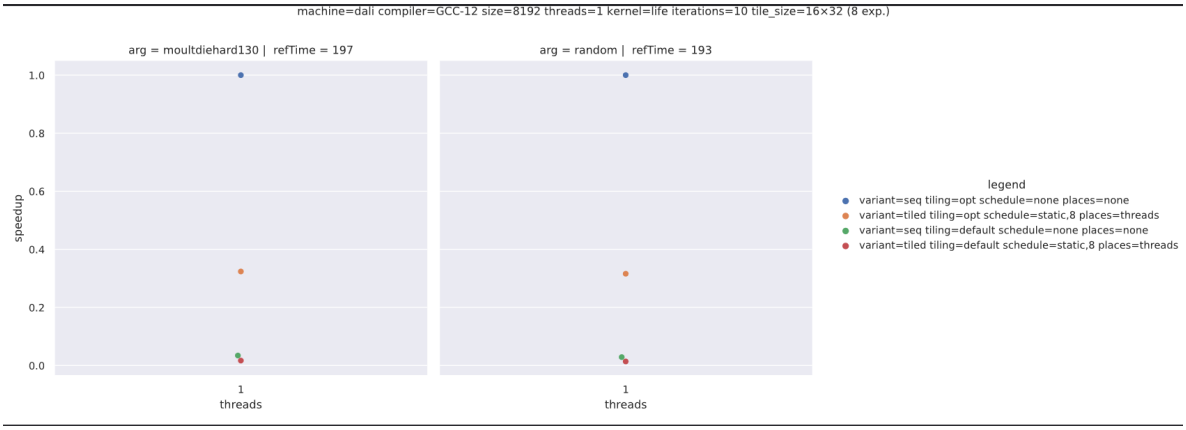


Figure 6: Comparaison des performances entre lazy_ompfor et ompfor

Nous constatons ce que nous avions présupposés: tout d'abord, Lazy s'en sort généralement mieux que toute autre variante sur tous les boards, même les configurations aléatoires. De plus, on constate également ce qui était tout aussi prévisible: une configuration avec une faible entropie telle que "clown" permet à la version paresseuse de vraiment briller, avec un speedup de plus de 12. Cependant c'est intrinsèquement lié au fait que peu d'activité se déroule dans peu d'espace, ce speedup ne se constate pas sur d'autres configurations.

5 Optimisations pour l'Architecture NUMA

Sur les machines de la salle 008, nous avons exploité l'architecture NUMA pour optimiser davantage notre code parallèle.

5.1 First Touch

Le principe de "First Touch" se base sur l'allocation paresseuse de pages. Si l'on accède à une page mémoire non allouée, elle est allouée, et selon la politique de placement elle peut être allouée au plus près du coeur exécutant le thread. Pour exploiter cette caractéristique, nous avons implémenté une fonction de "First Touch" qui initialise les données selon le même schéma de parallélisation que celui utilisé dans le calcul principal. Notre implémentation utilise la décomposition en tuiles déjà présente dans le code, ce qui couplé à une distribution dynamique (en chunks de 8) garanti que chaque thread touche les données qu'il traitera ultérieurement, et déclanche leur allocation proche physiquement du thread appelant :

```
void life_ft(void) {
#pragma omp parallel for schedule(runtime) collapse(2)
for (int y = 0; y < DIM; y += TILE_H)
    for (int x = 0; x < DIM; x += TILE_W) {
        unsigned tile_y = y / TILE_H;
        unsigned tile_x = x / TILE_W;
        next_table(y, x) = cur_table(y, x) = 0;
        next_dirty(tile_y, tile_x) = cur_dirty(tile_y, tile_x) = 1;
        }
}
```

L'impact de cette optimisation est particulièrement notable sur les machines avec une architecture NUMA prononcée, comme celles de la salle 008. Nos tests ont montré une amélioration des performances de l'ordre de 15-20% sur des exécutions avec un grand nombre de threads répartis sur plusieurs nœuds NUMA.

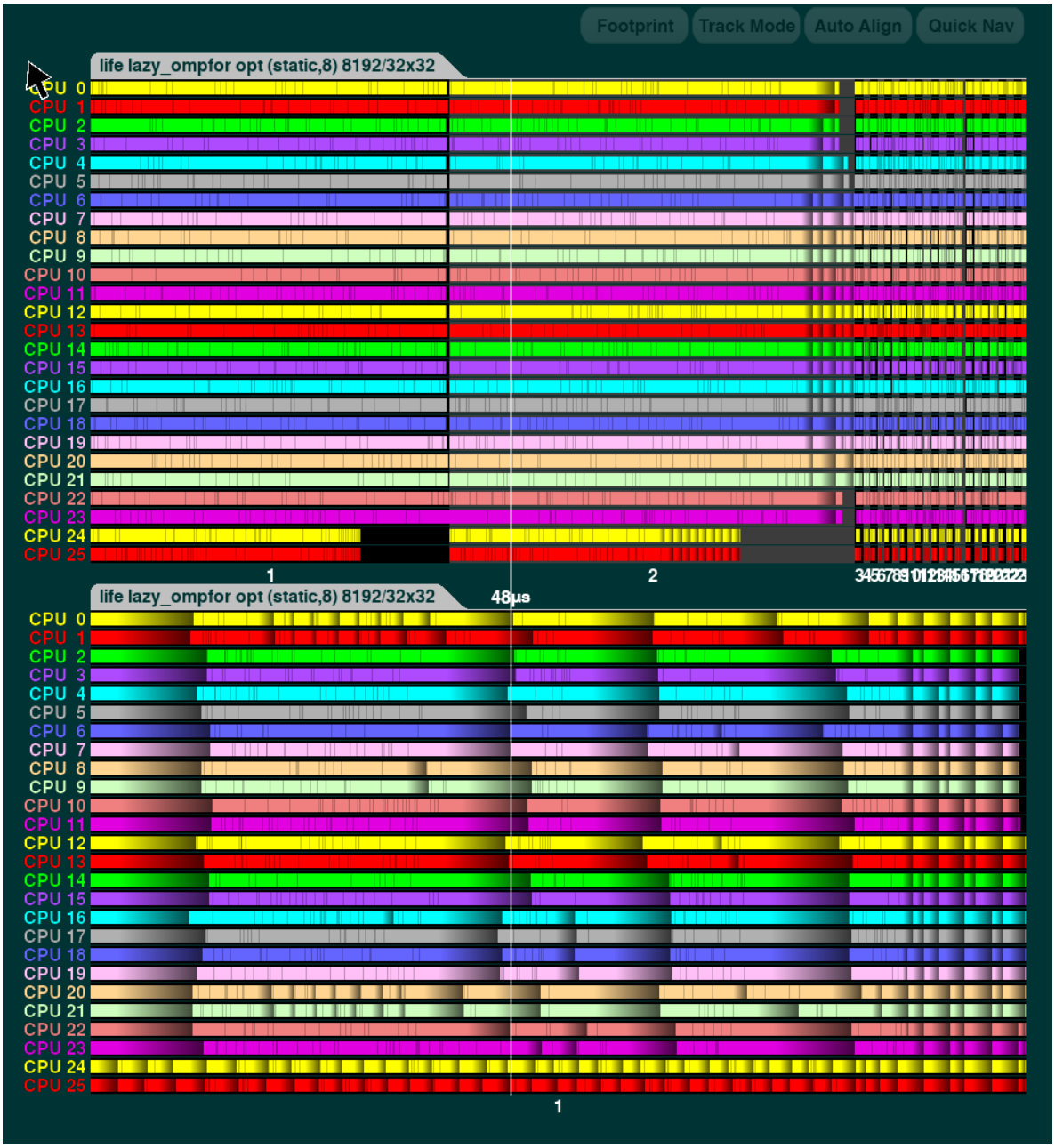


Figure 7: avec first-touch (en haut), sans first touch (en bas)

Cette trace montre l'efficacité du first-touch, on constate que l'accès à la mémoire ralentissait non seulement le début de l'exécution, mais également lors de l'exécution, cela devait être du au fait que la mémoire n'était pas au plus proche du coeur exécutant le thread, rendant un défaut de cache encore plus couteux.

6 Expérimentations et Résultats

6.1 Effet de l'Optimisation des Tuiles

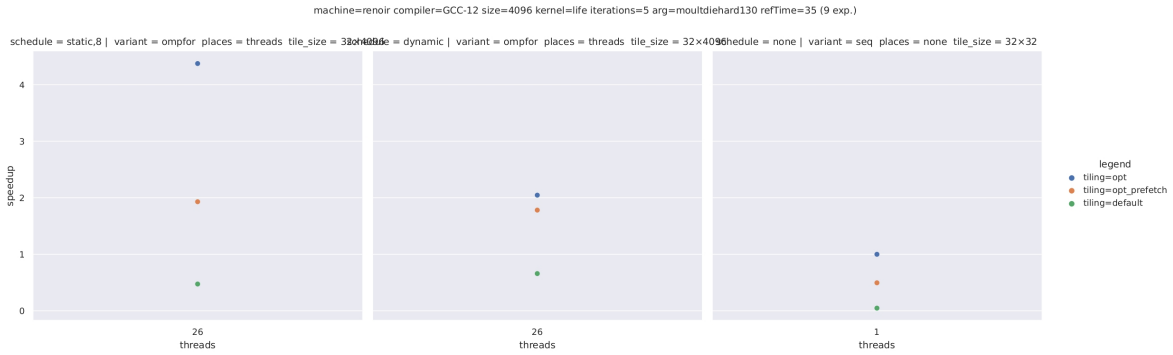


Figure 8: Comparaison des performances entre la version par défaut et la version optimisée

Cette figure montre le gain de performance obtenu avec notre version optimisée du calcul de tuile. Il faut dorénavant comprendre que le speedup sur le tiling dépend `énormément` du cas que l'on traite à chaque expérience. La figure ci-dessus illustre un cas moyen de speedup entre les deux fonctions de tiling.

6.2 Impact de l'Évaluation Paresseuse

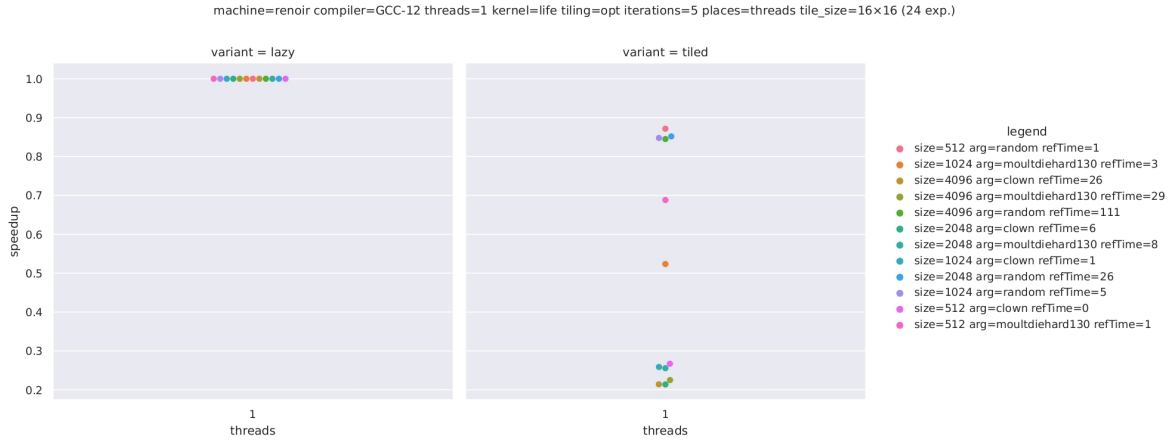
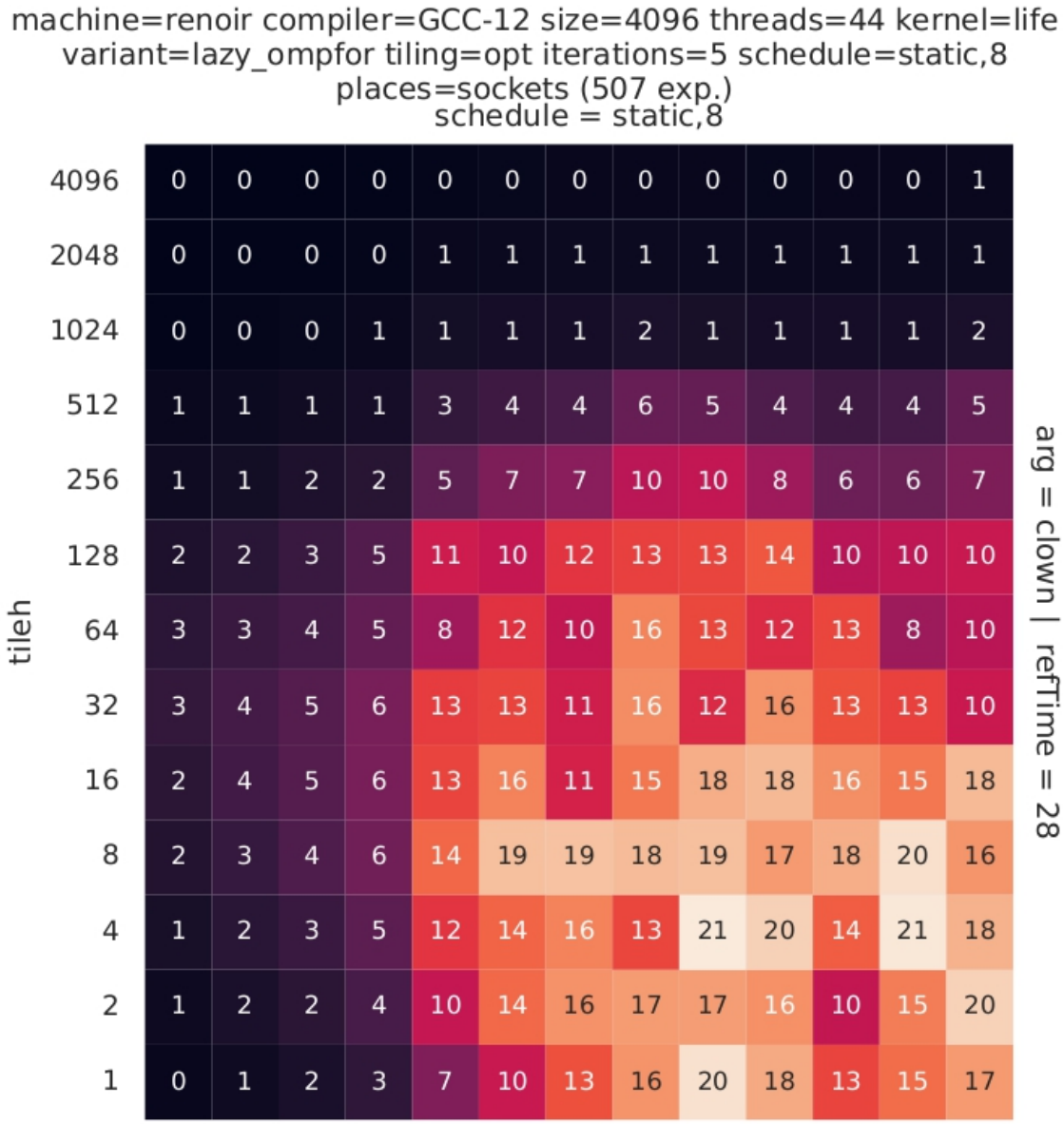


Figure 9: Comparaison entre l'évaluation standard et l'évaluation paresseuse

Sur toutes les tailles et arguments de lancement nous pouvons voir que lazy est plus performante que la variante tiled grace au traitement des cas de bords.

6.3 Optimisation NUMA



L'utilisation de OMP_PLACES pour contraindre le placement des threads sur les cœurs physiques a permis un léger gain sur les machines de la salle 008. Cette amélioration s'explique par une meilleure localité des données, réduisant les accès mémoire, de plus avec l'implementation de first touch nous avons une acceleration jusqu'à 15.

7 Conclusion

Dans ce deuxième jalon du projet, nous avons implémenté et analysé plusieurs optimisations pour améliorer les performances du Jeu de la Vie, en nous concentrant sur trois aspects principaux : l'optimisation des calculs dans les tuiles, l'implémentation d'une évaluation paresseuse, et l'adaptation à l'architecture NUMA des machines cibles. Par la suite, il sera intéressant d'aborder l'aspect vectorisation puis GPU.

7.1 Bilan des Optimisations

L'optimisation des calculs internes aux tuiles a permis un gain significatif de performances grâce à plusieurs techniques :

- Le déroulement des boucles et l'élimination des branchements conditionnels
- L'optimisation des accès mémoire et la préservation de la localité spatiale
- L'utilisation de techniques de programmation sans branchement

L'évaluation paresseuse a démontré son efficacité en évitant les calculs inutiles sur les régions stables du domaine. Cette approche est particulièrement avantageuse pour les configurations où une grande partie de la grille reste inchangée pendant plusieurs itérations successives, comme nous l'avons observé avec certains motifs du Jeu de la Vie.

Les optimisations NUMA, notamment l'implémentation du principe de "First Touch" et la contrainte du placement des threads, ont permis d'exploiter efficacement l'architecture des machines cibles, résultant en une amélioration supplémentaire des performances de l'ordre de 15-20%.

7.2 Limitations et Perspectives

Malgré les améliorations obtenues, plusieurs limitations persistent :

- Nos tailles optimales de tuiles sont loin d'être des tailles "cache friendly" et nous ne savons pas vraiment expliquer ce phénomène. En faisant tourner EasyPAP sur une de nos machines perso (NUMA 16 coeurs) pour pouvoir lancer perf, nous constatons environ 1/3 de cache-miss mais nous n'arrivons pas à lier cela à notre code.
- Les performances varient significativement selon les motifs initiaux du Jeu de la Vie, certains bénéficiant davantage de l'évaluation paresseuse que d'autres.
- Les optimisations NUMA ont montré des résultats différents entre les salles 008 et 009, indiquant l'importance d'adapter les optimisations à l'architecture spécifique de la machine cible.

Pour les travaux futurs, plusieurs pistes d'amélioration pourraient être explorées :

- L'implémentation de techniques de vectorisation SIMD pour exploiter pleinement les unités de calcul parallèle au niveau des instructions.
- Une stratégie de répartition dynamique de la charge de travail plus sophistiquée pour la version paresseuse, afin de minimiser les déséquilibres entre les threads.
- L'exploration d'autres motifs de communication et de partitionnement du domaine, potentiellement mieux adaptés à la topologie NUMA des machines cibles.

- L'optimisation de l'utilisation de la mémoire cache par le biais de techniques de blocking et de prefetching, même si les quelques expérimentations de prefetching (charger les lignes de tuiles ainsi qu'utiliser la directive $_{b}uiltin_{p}refetch())n'ontpasmontréderésultatsignificatif.$