ENSEIRB-MATMECA
Filière informatique

–Programmation Mulitcoeur et GPU–
–Semestre 8–

Optimisation Parallèle et GPU du Jeu de la Vie

Auteurs :
Pierre-Jean MOREL
Hugo Langlais

Encadrant :
M. Raymond NAMYST

31 janvier 2023

Table des matières

1 Introduction **2**

2 Versions Multicoeur **2**

2.1 Version de base 2

2.2 Optimisation paresseuse 4

2.3 Comparaison 7

3 Version GPU **8**

3.1 Version OpenCL de base 8

3.2 Version OpenCL paresseuse 9

4 Résultats expérimentaux **10**

4.1 Versions CPU 10

4.2 Versions GPU 12

5 Conclusion **12**

1 Introduction

Le jeu de la vie est un jeu à zéro joueur, se déroulant sur une grille à deux dimensions. Chaque case de la grille, appelée cellule, peut soit être morte, soit en vie. Le changement d'état se fait en fonction des 8 cellules voisines :

— Une cellule morte devient vivante si elle possède exactement trois voisins en vie, sinon elle reste morte

— Une cellule vivante doit posséder deux ou trois voisins en vie, sinon elle meurt.

On considère ici une simulation synchrone, toutes les cellules change d'état (en vie ou morte) en même temps, suivant l'itération précédente.

Le but de ce projet est de simuler le plus vite possible le jeu de la vie. Pour cela, nous devions exploiter autant que possible les capacités parallèles du matériel. Dans ce rapport, nous détaillons les optimisations réalisées, expliquons leur implémentation, ainsi que les gains de performance (ou non) obtenus.

Nous avons utilisé EasyPAP comme cadre de travail où une première version séquentielle du jeu nous est fournie. Les différentes courbes et expérimentations ont été réalisées sur les machines du CREMI (2x Xeon Gold 5118 64Go, NVIDIA RTX 2070, 8GB de RAM) ainsi que sur une de nos machines personnelles (Intel Core i5-11300H 3.10GHz 8 coeurs (4 physiques), NVIDIA GTX 1650, 8GB de RAM).

2 Versions Multicoeur

Dans un premier temps nous avons cherché à optimiser la version tuilée séquentielle en parallélisant le calcul de chaque tuile sur les différents processeurs disponibles.

2.1 Version de base

Une première version parallèle a été rapidement mise en place à l'aide d'OpenMP. Le principe est simple, il suffit d'attribuer le calcul de chaque tuile à un thread. De cette manière les différentes tuiles du jeu sont calculées en parallèle. Il faut également maintenant faire attention à protéger l'accès concurrent à la variable `change` (qui indique si une tuile a changé d'une itération à l'autre). Voilà comment cela se traduit avec OpenMP :

```
...

#pragma omp parallel for schedule(runtime) collapse(2) shared(change) private(temp)
for (int y = 0; y < DIM; y += TILE_H)
{
  for (int x = 0; x < DIM; x += TILE_W)
  {
    temp = do_tile (x, y, TILE_W, TILE_H, omp_get_thread_num());

    #pragma omp critical
    change |= temp;
  }
}

...
```

Listing 1 – Partie parallélisée avec OpenMP de la version `omp_tiled`

On remarque l'utilisation de la directive `for` pour paralléliser les boucles liées au calcul des tuiles couplé avec `collapse(2)` pour paralléliser les 2 boucles imbriquées. De plus, nous laissons `schedule` à `runtime` pour expérimenter une répartion statique ou dynamique des threads. On utilise également `atomic` au lieu de `critical` pour tirer profit du matériel.

Nous avons ensuite cherché à optimiser en utilisant différentes tailles de tuiles et différentes implémentations OpenMP. L'impact de la taille des tuiles sera discuté en partie 4

Par exemple, nous nous sommes demandé quelles étaient les différences d'utiliser `omp for` par rapport à une version avec une seule région parallèle et à base de `task` dont voici l'implémentation :

```
...
#pragma omp parallel shared(change, res) private(temp)
#pragma omp single
for (unsigned it = 1; it <= nb_iter; it++) {
  for (int y = 0; y < DIM; y += TILE_H)
  {
    for (int x = 0; x < DIM; x += TILE_W)
    {
      #pragma omp task
      {
        temp = do_tile (x, y, TILE_W, TILE_H, omp_get_thread_num());
        #pragma omp atomic
        change |= temp;
      }
    }
  }

  #pragma omp taskwait
  swap_tables ();
  ...
```

Listing 2 – Partie parallélisée avec OpenMP de la version `omp_task`

Cependant, la version basée sur les tâches est bien moins efficace comme on peut le voir sur la figure 1. Cela est probablement dû à l'attente des tâches qui demande une synchronisation entre les threads ainsi que la distribution des tâches et la gestion de l'exécution de la partie `single` par un seul thread. On remarque même que si le nombre de thread devient trop élevé (supérieur au nombre de coeurs) le surcoût de la parallélisation ne devient plus rentable et on devient moins rapide que la version séquentielle.

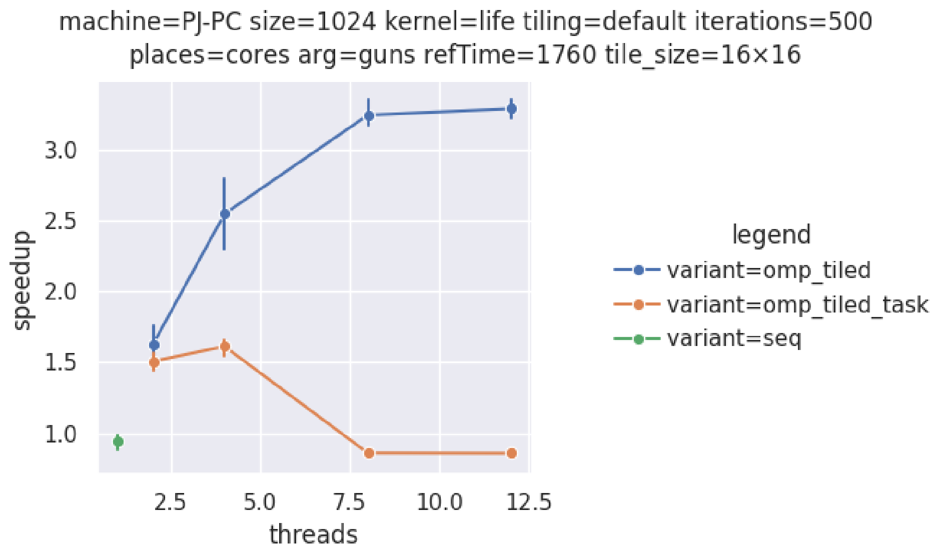

FIGURE 1 – Speedup en fonction du nombre de thread pour les version `omp_tiled` et `omp_tiled_task` (machine personnelle)

2.2 Optimisation paresseuse

Souvent dans ce jeu, on remarque qu'une grande partie de l'espace ne change pas et ne changera quasiment jamais. On peut alors naturellement se dire qu'il serait possible d'éviter de faire des calculs inutiles dans ces zones. Par exemple, sur la figure 2 (plus de 38 millions de pixels) les parties noires à l'intérieur des cellules pourraient ne pas être considérées pour gagner en temps de calcul et libérer des threads pour mieux répartir la charge.

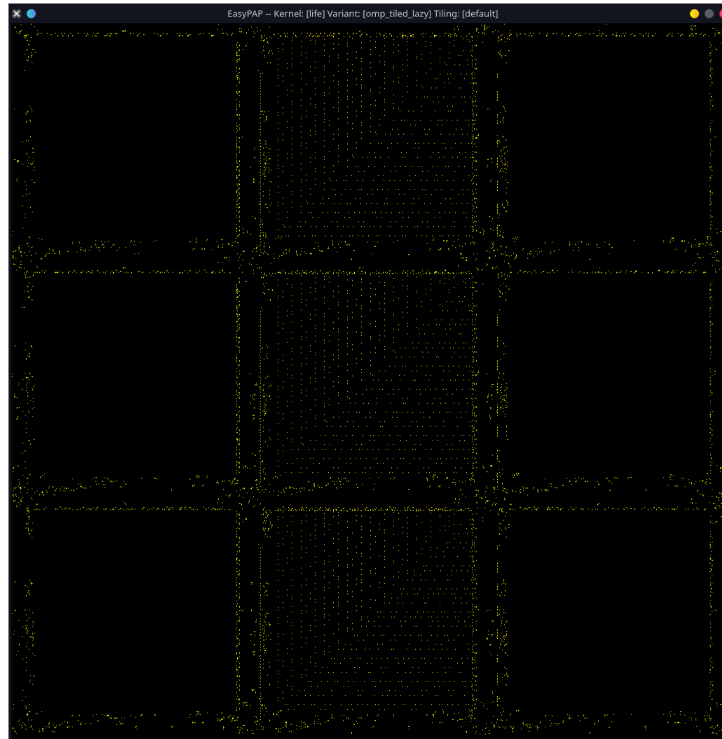

FIGURE 2 – Exemple de configuration meta3x3 avec beaucoup de zones inchangées

La version paresseuse se base sur le principe suivant : L'état d'une cellule est inchangé si aucun de ses 8 voisins n'a changé. Appliqué à une cellule, nous avons toujours besoin de vérifier les 8 voisins, mais si on applique ce prince aux tuiles, c'est le calcul de toute la tuile qui peut être évité. On effectue donc cette vérification en se servant de deux tableaux qui représentent le changement de l'ancien et du prochain (`last_changed_table` et `next_changed_table` état pour chaque tuile :

```
unsigned tiles_around_changed(int x, int y)
{
  if (last_changed_table(y, x))
    return 1;

  unsigned changed = 0;

  for (int yloc = y - 1; yloc < y + 2; ++yloc)
    for (int xloc = x - 1; xloc < x + 2; ++xloc)
      if (yloc >= 0 && yloc < (DIM / TILE_H) && xloc >= 0 && xloc < (DIM / TILE_W))
        changed |= last_changed_table(yloc, xloc);

  return changed;
}

...
    temp = tiles_around_changed(x / TILE_W, y / TILE_H);

    if (temp)
      temp = do_tile (x, y, TILE_W, TILE_H, omp_get_thread_num());
```

```
    next_changed_table(y / TILE_H, x / TILE_W) = temp;
...
```

Listing 3 – Optimisation paresseuse OpenMP `omp_tiled_lazy`

Le plus important pour cette optimisation est de choisir une taille de tuile adaptée. En effet, si une tuile est modifiée, toutes les tuiles voisines doivent être recalculées. Choisir des tuiles plus petites permet de réduire la surface du plateau à recalculer. D'un autre côté, si les tuiles sont trop petites, la charge de travail entre les threads est mal répartie. Il faut donc trouver un équilibre.

La figure 3 illustre ce propos avec des tailles de tuiles de taille 4 et 8. Chaque tuile est délimitée en rouge. Le coin supérieur droit est modifié en noir (naissance d'une cellule). Les cellules vérifiées sont en vert. Pour des tuiles de taille 8x8, on doit vérifier/mettre à jour les 256 cellules, seulement 64 pour la version 4x4. Notons que dans notre exemple, on considère uniquement 3 des 8 tuiles voisines.

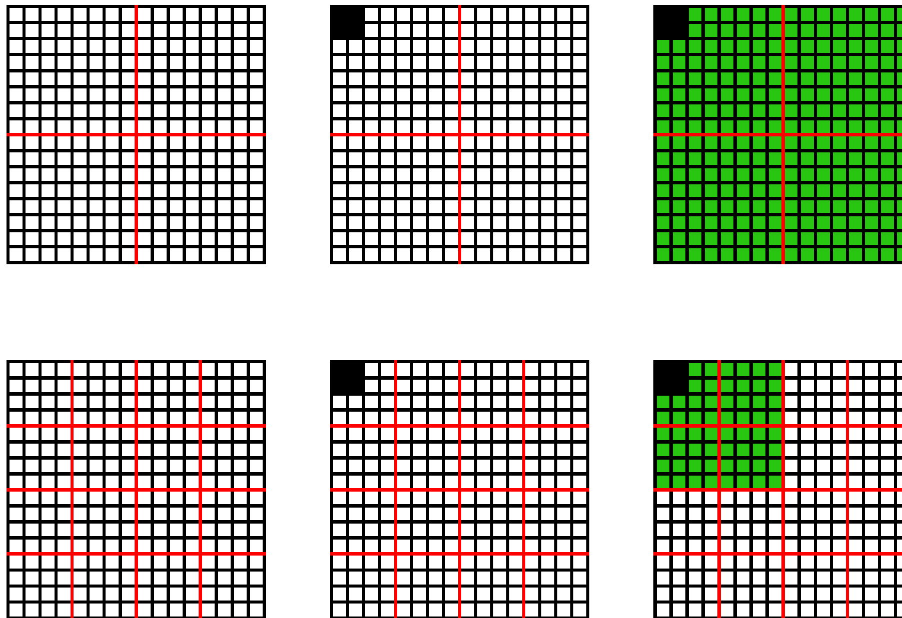

Figure 3 – Exemple de vérification/mise à jour pour tes tuiles 4x4 et 8x8.

Il n'existe pas de taille de tuile optimale pour tout jeu. La taille optimale dépend de la répartition des changements d'état sur le plateau à un instant donné. Nous avons donc pensé à trois manières d'obtenir une taille de tuile optimale :

- Pour chaque type de jeu, on utilise plusieurs tailles de tuiles et on choisit celle qui s'adapte mieux. Notons que cela n'est pas possible sur un jeu aléatoire.
- Faire évoluer la taille des tuiles au cours du jeu : En fonction du pourcentage de tuiles modifiées, on peut déterminer s'il faut agrandir ou non leur taille. Il faut cependant définir des bornes, pour pouvoir répartir la charge entre les threads.
- Faire évoluer la taille des tuiles au cours du jeu et en fonction les zones "actives" : Une grande zone sans activité cellulaire peut être regroupés en une seule tuile et ainsi être traitée extrêmement rapidement. Au contraire, une zone avec plus de changements d'états doit être coupée en tuiles de petite taille pour mieux s'adapter au profil des cellules.

L'évolution de la taille des tuiles au cours du jeu peut sembler intéressante, mais elle demande des calculs supplémentaires. Sa pertinence est donc à mesurer. Par souci de temps et de compétences, nous avons donc choisi une taille de tuiles fixe.

2.3 Comparaison

En pratique, voilà les résultats de speedup en figure 8 qu'on obtient pour `omp_tiled` et `omp_tiled_lazy` sur la configuration de la figure 2.

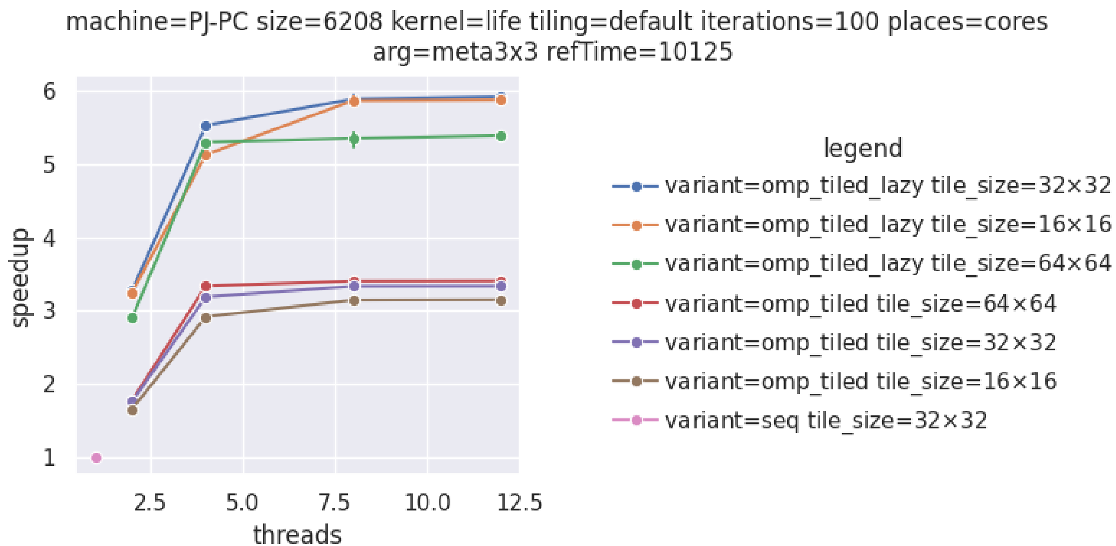

FIGURE 4 – Speedup en fonction du nombre de threads et de la taille des tuiles pour les version `omp_tiled` et `omp_tiled_lazy` sur la configuration meta3x3 (sur machine personnelle)

On observe un pic de performance pour des tuiles moyennes de 32x32 comme expliqué précédemment et on observe également un gros gain sur cette configuration grâce à l'optimisation paresseuse (on va presque deux fois plus vite !). Il est également assez intéressant de voir en figure 5 la proportion d'image qui n'est pas complètement traitée, car on rappelle qu'on effectue quand même un calcul de voisinage qui force le thread à effectuer du travail même s'il en fait très peu on accapare les ressources pour un court instant ce qui explique que le speedup n'est pas proportionnel à la surface de l'image qui n'est pas calculé.

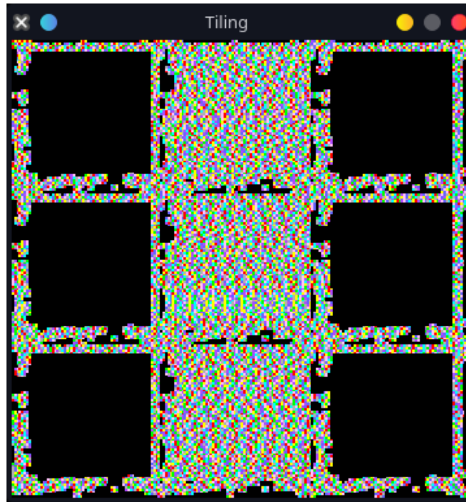

FIGURE 5 – Répartition du travail pour les threads sur meta3x3 avec `omp_tiled_lazy` et des tuiles 32x32

3 Version GPU

Une fois nos versions sur processeur optimisées, nous avons voulu développer une version pour carte graphique afin d'utiliser toute la puissance de calcul disponible.

3.1 Version OpenCL de base

Le jeu de la vie est très adapté à la parallélisation. Le nombre élevé de threads sur GPU nous permet de jouer bien plus rapidement que sur processeur en affectant tous les threads disponibles à une tuile. Cette version est très simple, il suffit de garder à l'esprit que sur GPU nous avons un thread par pixel de l'image. Étant donnée que nous avons une image de sortie et d'entrée, il n'y pas non plus de synchronisation à effectuer. Voilà le kernel OpenCL correspond :

```
__kernel void life_ocl (__global unsigned *in, __global unsigned *out)
{
  int x = get_global_id (0);
  int y = get_global_id (1);

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

    for (int yloc = y - 1; yloc < y + 2; yloc++)
      for (int xloc = x - 1; xloc < x + 2; xloc++)
        n += in[yloc * DIM + xloc];

    n = (n == 3 + me) | (n == 3);

    out[y * DIM + x] = n;
  }
}
```

Listing 4 – Kenel OpenCL du jeu de la vie

On récupère donc les coordonnées globales du thread sur l'image et nous pouvons directement accéder le tableau avec ces indices.

3.2 Version OpenCL paresseuse

Vu l'efficacité de la version multicoeur paresseuse, nous étions curieux de savoir si une version paresseuse sur GPU serait également très efficace. Nous avons alors modifié le kernel précédent pour effectuer la vérification du changement des tuiles voisines. Nous avons tout d'abord alloué deux nouveaux buffers sur la carte correspondant à l'état des tuiles comment sur la version CPU. Ensuite, on effectue la vérification avec un seul thread pour ne pas faire le calcul plusieurs fois, et on vérifie également que les accès au prochain tableau de changement des tuiles sont atomiques car tous les thread d'un même groupe y accède en même temps.

```
__kernel void life_ocl_lazy (__global unsigned *in, __global unsigned *out, __global unsigned *last_changed
{
    int tile_x = get_group_id (0);
    int tile_y = get_group_id (1);
    int xloc = get_local_id (0);
    int yloc = get_local_id (1)

    // Verification de la tuile par le premier thread
    local unsigned changed

    if (xloc == 0 && yloc == 0) {
      changed = 0
      for (yloc = tile_y - 1; yloc < tile_y + 2; ++yloc)
        for (xloc = tile_x - 1; xloc < tile_x + 2; ++xloc)
          if (yloc >= 0 && yloc < (GPU_SIZE_Y / GPU_TILE_H) && xloc >= 0 && xloc < (GPU_SIZE_Y GPU_TILE_W))
            changed |= last_changed[yloc * (GPU_SIZE_Y / GPU_TILE_H) + xloc]
      changed |= last_changed[tile_y * (GPU_SIZE_Y / GPU_TILE_H) + tile_x];
    }

    barrier (CLK_LOCAL_MEM_FENCE);

    if (!changed) {
      return;
    }

    ...

    // Ecriture du nouvel etat de la tuile par tous les threads
    volatile __global unsigned* changed_ptr = next_changed + tile_y * (GPU_SIZE_Y / GPU_TILE_H) + tile_x;
    atomic_or(changed_ptr, n != me);
}
```

Cependant, vu que le kernel est exécuté par tout le monde, on se gagne pas et on perd même des performances avec cette version. Il faudrait trouver un moyen de lancer le groupe de thread en vérifiant au préalable la tuile pour ne pas lancer tous les threads et les faire attendre inutilement si aucun changement n'a au final été détecté. Ici tous les threads débutent le kernel et se stoppent sur la barrière.

Remarque Nous n'avons pas réussi à effectuer des dumps corrects pour les versions ocl sûrement dû à un buffer non mis à jour, en effet ils ne correspondait pas aux images finales que l'on observait à la fin.

4 Résultats expérimentaux

Afin de valider la simulation de nos versions du jeu de la vie, nous avons réalisé un script `validate_life` qui étant donné un nom de version, la lance sur différents nombres d'itérations et vérifie avec la commande `diff` que le résultat correspond avec les images précalculées avec la version séquentielle. Comme le jeu de la vie est déterminé uniquement par l'état initial des cellules, chaque version doit pouvoir générer exactement l'image de référence. Nos images de références sont générés avec la version séquentielle donnée comme base du projet. Les paramètres des images sont les suivants :

Taille de l'image	Itération
2176	500
2176	1000
2176	5000
2176	50000
64	100
512	100
1024	100

Chaque version optimisée a été comparée à la version séquentielle. Pour cela, nous utilisons comme unité le `speedup`, qui est le rapport entre le temps nécessaire de la version optimisée et la version séquentielle. Si `speedup` vaut 2, l'exécution prend 2 fois moins de temps avec la version optimisée.

4.1 Versions CPU

Nous avons commencé par comparer les versions CPU optimisées avec la version séquentielle. Tout d'abord, nous avons cherché à optimiser la taille des tuiles. Nous obtenons les résultats suivants figure ??.

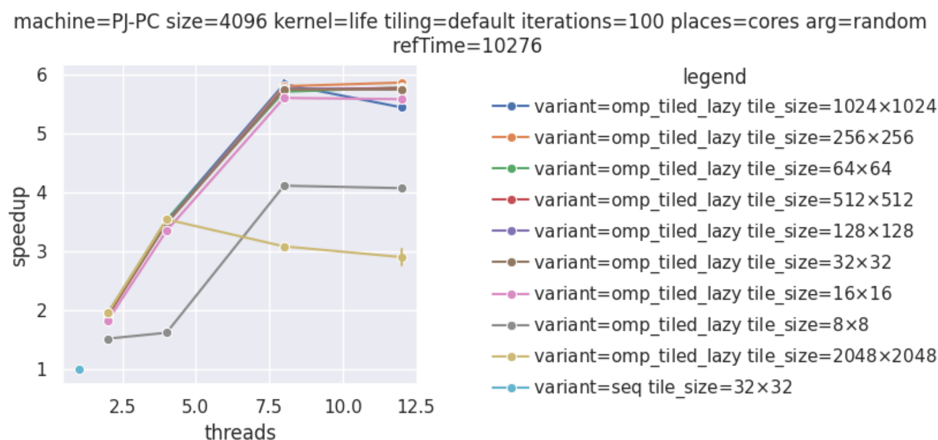

FIGURE 6 – Influence de la taille des tuiles et du nombre de threads sur la version `omp_tiled_lazy`

Nous voyons que la parallélisation est très efficace. On voit aussi que plus les tuiles sont grandes, plus le programme est rapide. Attention tout de même, des tuiles top grandes se rapprochent de la version séquentielle. Il faut donc trouver un équilibre. Pour la suite, nous avons choisi une taille de tuile de 32x32.

Nous avons également testé des tuiles rectangulaires, les résultats étaient identiques aux tuiles de même taille mais carrés.

Une fois la taille des tuiles optimisée, nous avons ensuite comparé les versions opm en jouant sur le nombre de threads, le nombre d'itérations et la taille de l'image. Nous obtenons les résultats figure 7 et 8.

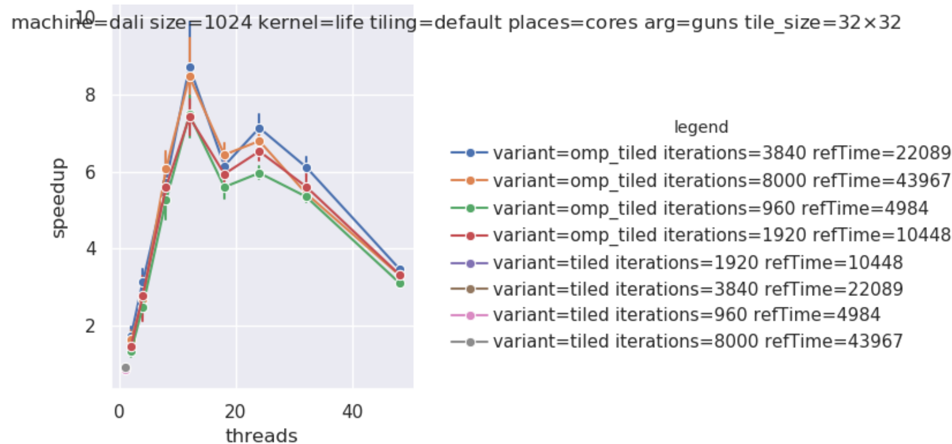

FIGURE 7 – Speedup pour la versions OpenMP de base en fonction du nombre de threads, des itération et de la taille de l'image

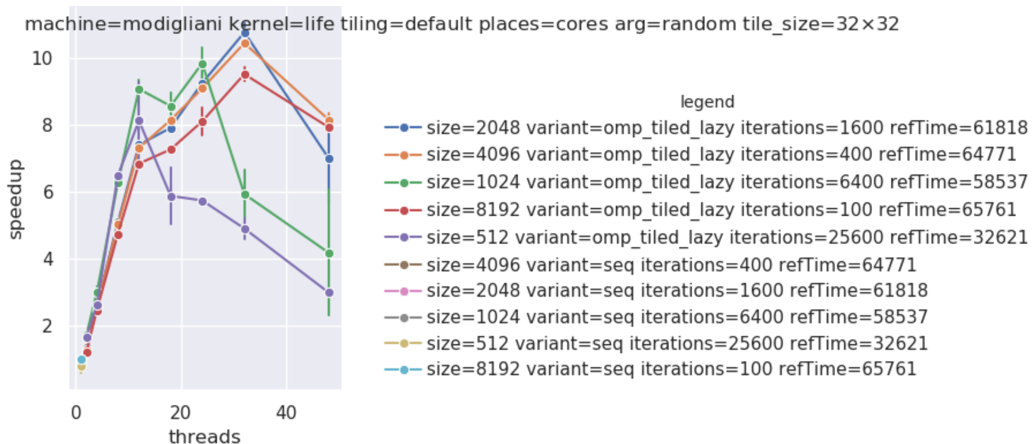

FIGURE 8 – Speedup pour la versions OpenMP paresseuse en fonction du nombre de threads, des itération et de la taille de l'image

Les performances baissent quand le nombre de threads devient trop grand. Nous pensons que cela est dû à une mauvaise répartition des tâches quand le nombre de thread est trop important. Le coût de création de tâche devient plus grand que le gain de la parallélisation car les threads n'ont pas assez de tâche. En effet, sur une image de taille 1024 avec des tuiles de taille 32, on se rend bien compte que plus de 24 threads sont inutiles. C'est notamment pourquoi on observe une chute en 8 pour des tailles 1024 et 512 et 24 threads alors que c'est un pic maximal pour les tailles supérieures.

4.2 Versions GPU

Nous avons ensuite comparé nos versions GPU avec la version séquentielle. Les résultats obtenus figure 9 montrent que la parallélisation GPU est bien plus efficace que celle faite par le CPU.

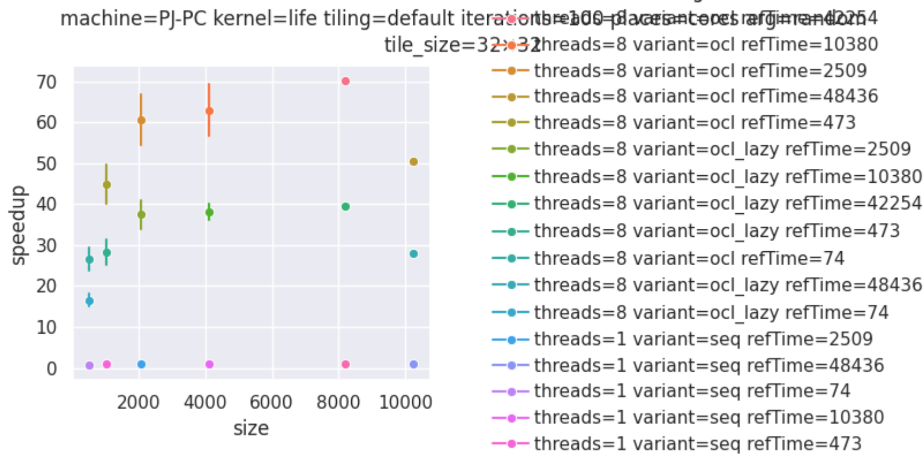

FIGURE 9 – ocl

Nous aurions aimé voir l'influence de la taille de tuile sur la version GPU. Malheureusement, EasyPAP ne permet d'utiliser uniquement des tailles de tuiles 16x16 malgré le fait de changer le paramètre `-ts`. On remarque cependant clairement le surcoût de la version paresseuse du au non déterminisme du parallélisme sur GPU.

5 Conclusion

Ce projet nous a permis de mettre en application les différentes méthodes d'optimisation vues en cours et en TP. Il pousse également à analyser le comportement des machines afin de comprendre les résultats obtenus. L'application au jeu de la vie à rendu le projet très ludique, et il était satisfaisant de le voir se dérouler de plus en plus vite, jusqu'à 70 avec la version GPU !

Malgré le fait d'avoir réalisé une version OMP, OCL et paresseuse pour les deux, nous aurions également aimé pouvoir expérimenter la vectorisation ou différentes optimisations GPU avec plus de temps pour tirer complètement profit des capacités des machines modernes.