Rapport GPU

Jeu de la vie

Encadrant

Raymond Namyst

Équipe

Jonathan Aucourt
Lucile Thiénot

Table des matières

Introduction

L'objectif de ce projet est de comparer les performances de plusieurs implémentations du jeu de la vie. Le jeu de la vie simule une forme de vie très basique : une cellule doit avoir des voisins pour vivre, mais elle meurt si elle en a trop. Plus précisément, les règles du jeu de la vie sont les suivantes :

— une cellule morte devient vivante si elle a exactement trois cellules voisines vivantes, sinon elle reste morte,

— une cellule vivante reste vivante si elle a deux ou trois cellules voisines vivantes, sinon elle meurt.

Nous avons fait des comparaisons de performances sur des implémentations séquentielles, OpenMP et OpenCL de ce jeu.

1 Implémentation

1.1 Versions séquentielles

1.1.1 Version 0

La version séquentielle basique consiste en une simple imbrication de boucles `for` permettant de parcourir linéairement l'intégralité de l'image et vérifiant pour chaque cellule si son nombre de voisins lui permet de vivre. Si aucune cellule n'a changé d'état, le nombre d'itérations effectué est retourné.

1.1.2 Version 1

La première version tuilée permet non pas de parcourir l'image linéairement mais de la parcourir "tuile par tuile". Autrement dit, on commence par parcourir la portion de l'image de taille `TILE_SIZE` par `TILE_SIZE` située dans le coin haut gauche de l'image, puis le carré juste à sa droite, etc.

1.1.3 Version 2

La version séquentielle tuilée optimisée permet de parcourir uniquement les tuiles susceptibles de subir un quelconque changement dans l'état de leurs cellules. En effet, on remplit à chaque tour deux tableaux de booléens :

— le premier permet de décider quelles briques ont subi un changement à l'itération précédente,

— le second permet de décider quelles briques ont subi un changement sur un de leurs bords lors de l'itération précédente.

Ainsi, une brique qui n'a pas été modifiée au tour précédent et dont les bords des voisins n'ont pas été modifiés ne sera pas recalculée : aucune de ses cellules n'a de raison de changer d'état.

Notons qu'une fonction d'initialisation des tableaux a été implémentée et est appelée depuis le programme principal.

1.2 Versions openMP `for`

1.2.1 Version 3

La version numéro 3 correspond à la version 0, à laquelle à été ajouté un `#pragma omp parallel for collapse(2) shared(img_stable)` concernant les `for` imbriqués.

La variable `img_stable` est un booléen permettant de décider si l'image s'est ou non stabilisée. Dans ces boucles `for` parallélisées, elle ne peut qu'être passée à 0 par des threads ayant au préalable modifié l'état de l'image. Cette variable ne nécessite donc pas de protection particulière et est bien sûr partagée par les threads.

1.2.2 Version 4

La version numéro 4 correspond à la première version tuilée, à laquelle a été ajouté un `#pragma omp parallel for collapse(2) shared(img_stable)` concernant les `for` imbriqués principaux, autrement dit ceux permettant de passer de tuile en tuile : les threads se partagent donc les différentes tuiles.

1.2.3 Version 5

La version 5 est l'équivalent de la version précédente, mais pour la version tuilée optimisée. Les threads se partagent donc les tuiles, mais certaines ne sont en réalité pas recalculées.
Ceci est notre version openMP la plus rapide.

1.3 Versions openMP `task`

1.3.1 Version 6

On utilise ici des tâches openMP pour paralléliser la version 1. Ainsi, un thread unique va créer des tâches à exécuter par les autres threads, avant que ceux-ci ne les exécutent. Une tâche correspond ici à une tuile.

Les variables i et j sont définies comme `firstprivate`, car chaque valeur du couple (i, j) est associée à une tuile et est utilisée lors du parcours de la tuile. Il est donc nécessaire qu'un thread exécutant une tâche en ait une copie privée.

1.3.2 Version 7

Cette version est l'équivalent de la version précédente, mais appliquée à la version tuilée optimisée (v2).

1.4 Versions openCL

1.4.1 Version 8 (kernel `life_naif`)

Dans cette version naïve du jeu de la vie, chaque thread se contente de compter ses voisins vivants avant de décider si la cellule dont il s'occupe peut ou non vivre dans cet environnement. Chaque thread fait donc huit accès mémoires en plus de celui lui permettant de changer la valeur de sa cellule.

1.4.2 Version 9 (kernel `life`)

Chaque thread contribue désormais au remplissage de la tuile dont il fait partie. Cette tuile est plus large que celles utilisées en openMP : elle comprend également les bords extérieurs de la tuile, le parcours de la tuile nécessitant la connaissance de l'état des cellules présentes autour d'elle.

Ensuite, chaque thread se réfère à la tuile ainsi remplie pour compter ses voisins vivants et définir son état.

Avec cette méthode, en moyenne un seul accès mémoire est effectué par cellule de l'image, ces cellules étant ensuite stockées localement. Seules les cellules se trouvant à la frontière de plusieurs tuiles sont accédées plusieurs fois, autrement dit lors du calcul de chacune de ces tuiles.

Notons qu'une version 10 existe dans le code, et utilise le kernel `life_stop`. Cette version utilise un booléen qui permet de stopper le programme lorsque l'image est stabilisée. Cependant, le grand nombre de vérifications que ce système implique diminue fortement les performances. Cette version ne présente donc que peu d'intérêt et n'est pas utilisée lors des comparaisons de performances.

2 Comparaison de performances

Nous allons maintenant comparer les performances des différentes versions que nous avons implémentées.

2.1 Comparaison des versions séquentielles

La courbe 1 compare les performances des versions séquentielles du jeu de la vie sur une configuration `guns` de taille 1024 avec des tuiles de taille 32.

La version séquentielle de base (v0 en noir) et la version séquentielle tuilée (v1 en rouge) ont des performances équivalentes, ce qui est normal puisqu'en séquentielle, tuilée ou non, il faut parcourir l'ensemble de la grille. La v1 est moins bonne que la v0 car l'accès aux données se fait de façon contigue pour la v0 et non pour la v1. La version tuilée optimisée (v2 en vert) est meilleure car elle ne recalcule pas les tuiles qui ne changent pas.

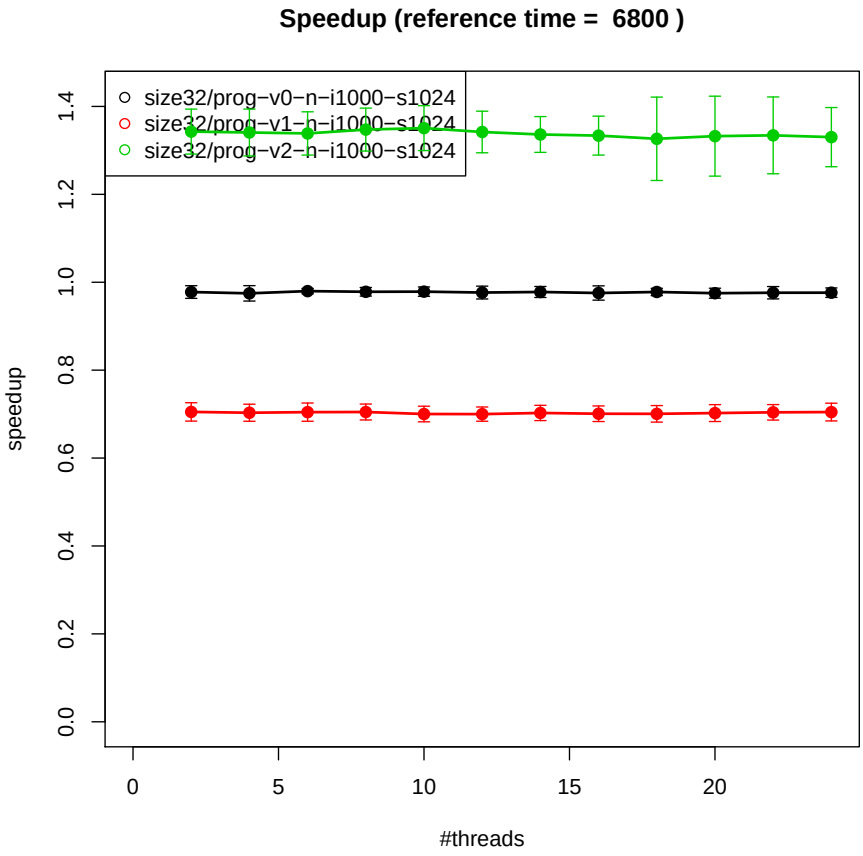

FIGURE 1 – Versions séquentielles

2.2 Comparaison des versions `OpenMP for`

La courbe 2 compare les performances des versions `OpenMP for` sur une configuration `guns` de taille 1024 avec des tuiles de taille 64. Notons que l'ordonnancement est statique, l'ordonnancement dynamique étant utilisé en partie 2.10.

La version de base (v3 en noir) est moins performante que la version tuilée (v4 en rouge), elle-même moins performante que la version tuilée optimisée (v5 en vert). Nous pouvons remarquer que les trois versions `OpenMP for` sont plus efficaces lorsque le nombre de threads est égal à 8. Cela doit correspondre à un stade où `OpenMP` est capable de répartir le plus efficacement les 256 tuiles sur les différents threads.

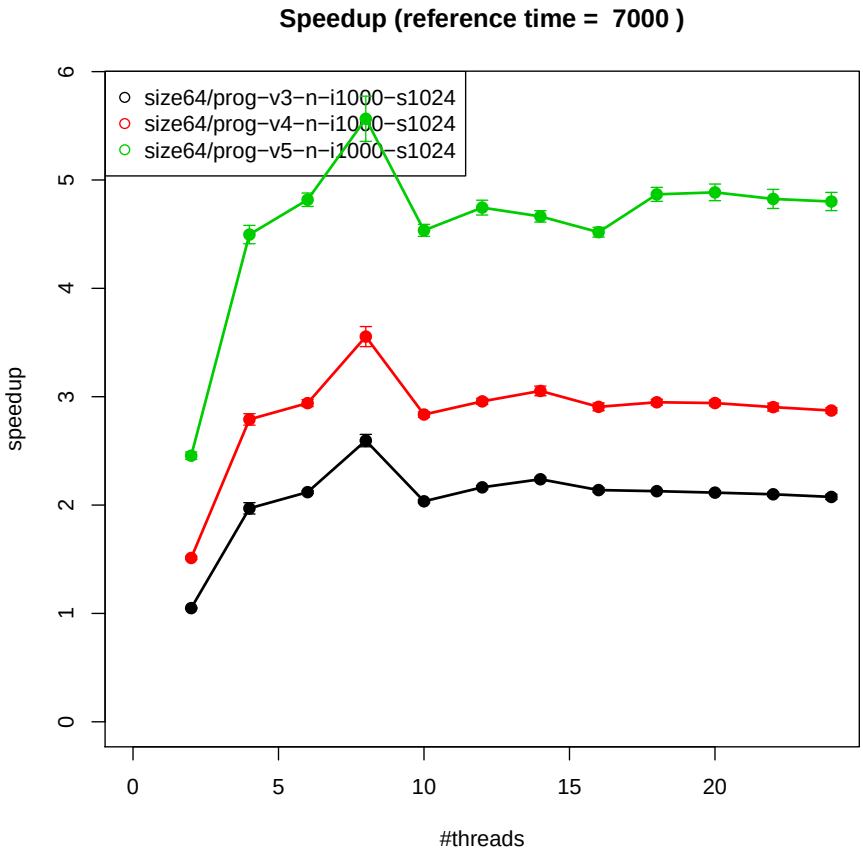

FIGURE 2 – Versions `OpenMP` for

2.3 Comparaison des versions `OpenMP task`

La courbe 3 compare les performances des versions `OpenMP task` sur une configuration `guns` de taille 1024 avec des tuiles de taille 64.

La version tuilée (v6 en noir) est moins performante que la version tuilées optimisée (v7 en rouge).
En effet, ce programme a été exécuté sur une image de base avec lanceurs : un certain nombre des tuiles de cette image sont (le plus souvent) stables, et ne nécessitent donc que rarement d'être recalculées. Ceci explique les meilleures performances de la version tuilée optimisée avec `OpenMP task`.

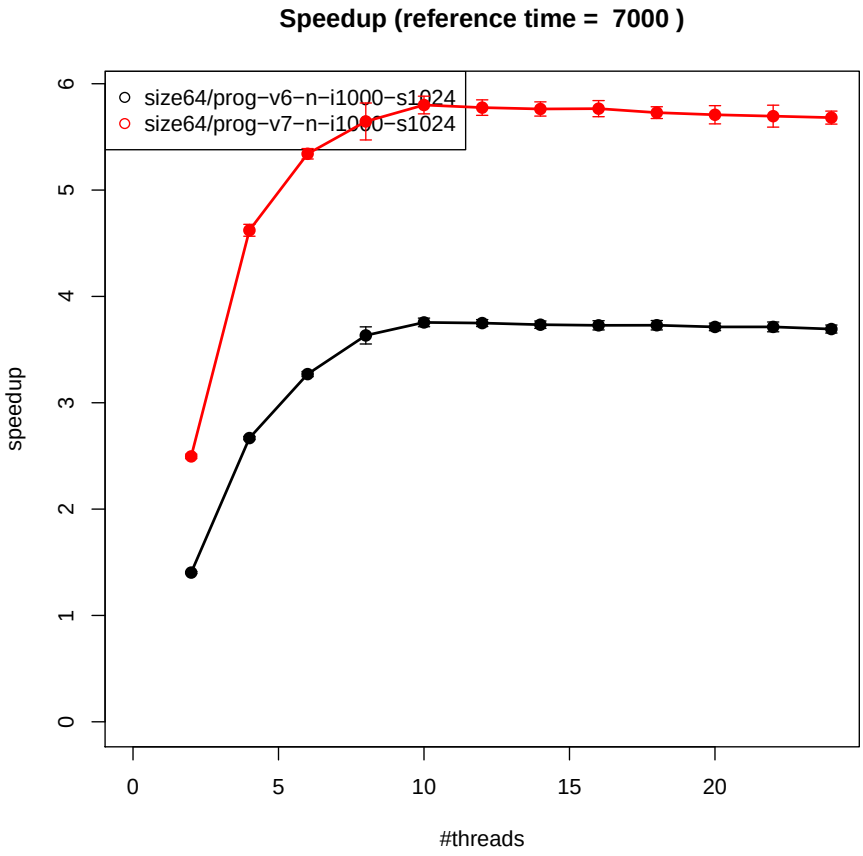

FIGURE 3 – Versions `OpenMP task`

2.4 Comparaison des versions `OpenCL`

La courbe 4 compare les performances des versions `OpenCL` sur 10000 itérations, une configuration aléatoire de taille 4096 et des tuiles de taille 16. La version naïve (v8 en noir) est légérement moins performante que la version non naïve (v9 en rouge). Cependant, les performances entre les deux versions sont très similaires.

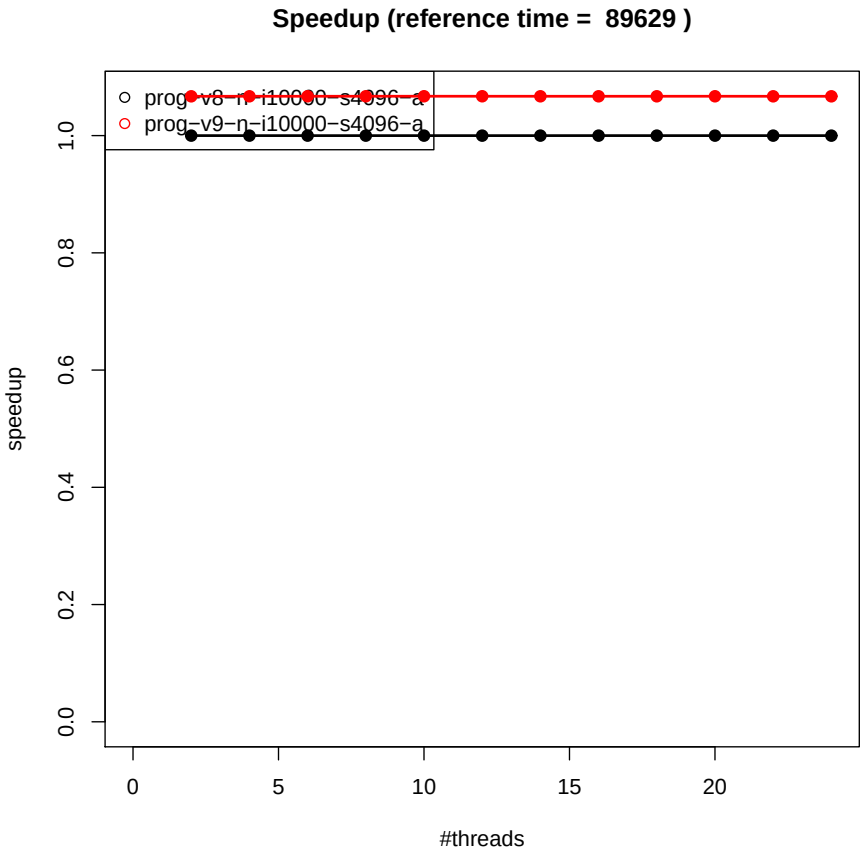

FIGURE 4 – Versions `OpenCL`

Une comparaison de l'exécution de la version optimisée avec des tuiles de taille 16 et avec des tuiles de taille 32 nous a montré que les performances ne variaient en l'occurrence pas en fonction de la taille des tuiles.

2.5 Comparaison des versions de base

La courbe 5 compare les performances des versions de base séquentielle et `OpenMP for` sur une configuration `guns` de taille 256. La version séquentielle (v0 en noir) est, sans grande surprise, beaucoup moins performante que la version `OpenMP for` (v3 en rouge) qui répartit le travail équitablement entre les différents threads. La version parallèlisée est donc plus performante que la version séquentielle, ce qui est rassurant.

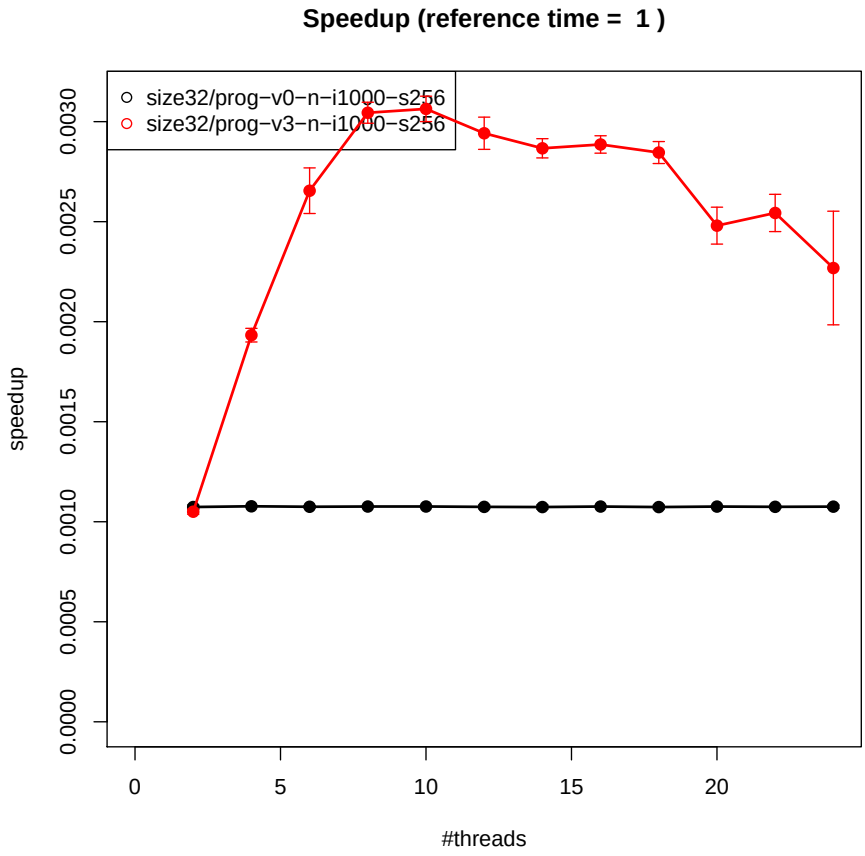

FIGURE 5 – Versions de base

2.6 Comparaison des versions tuilées

La courbe 6 compare les performances des versions tuilées séquentielle, `OpenMP for` et `OpenMP task` sur une configuration `guns` de taille 256 avec des tuiles de taille 32.

La version séquentielle (v1 en noir) est moins performantes que les deux autres (ce qui est normal, la version séquentielle est moins bonne que les versions parallèlisées). La version `OpenMP for` (v4 en rouge) est meilleure que la version `OpenMP task` (v6 en vert). Cela est probablement dû au temps de création des tâches durant lequel un seul thread est actif (toutes les tâches sont créées par un seul thread).

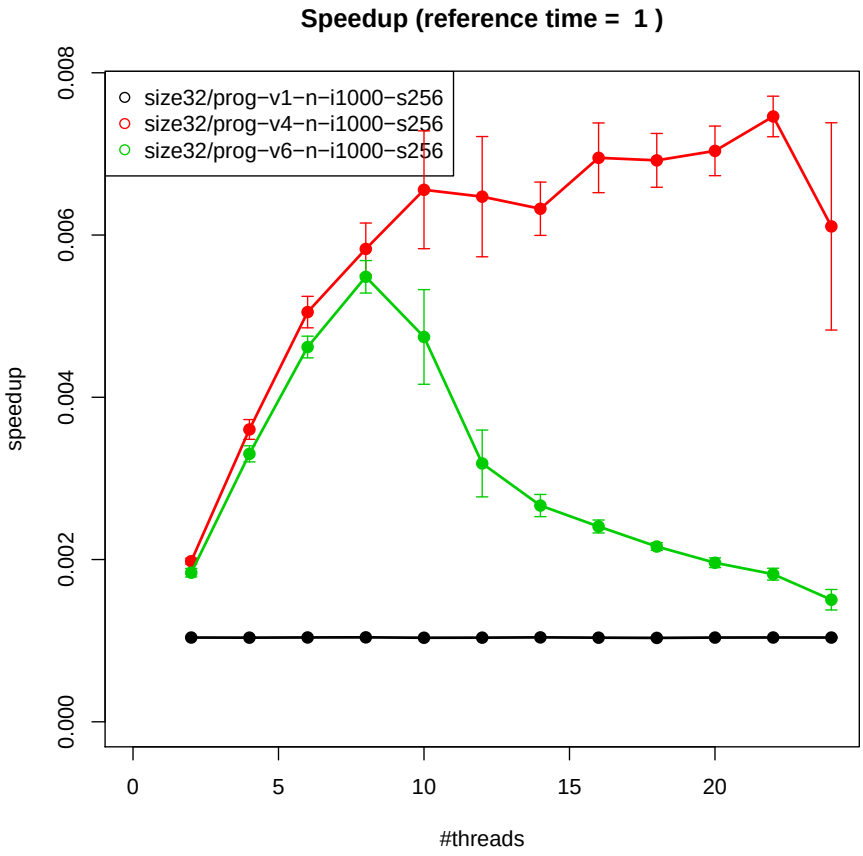

FIGURE 6 – Versions tuilées sur une image de taille 256 et de tuiles 32

Cependant, comme on peut le constater en figure 7, la tendance est bien différente sur des images de taille 1024 et de tuiles 64 par exemple : la courbe rouge représente alors la version `OpenMP task`, qui est rapidement meilleure que la version `OpenMP for` (en noir). Nous reparlerons de cette différence dans la partie suivante, concernant les versions tuilées optimisées.

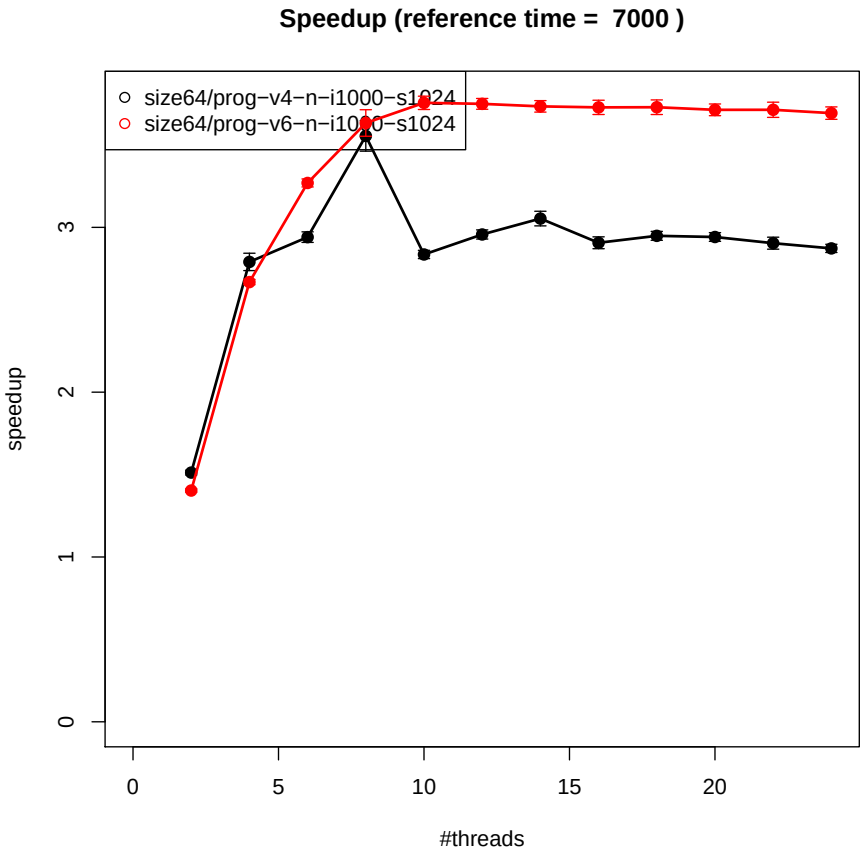

FIGURE 7 – Versions tuilées sur une image de taille 1024 et de tuiles 64

2.7 Comparaison des versions tuilées optimisées

La courbe 8 compare les performances des versions tuilées optimisées séquentielle, `OpenMP for` et `OpenMP task` sur une configuration `guns` de taille 256 avec des tuiles de taille 32. La version séquentielle (v2 en noir) est moins performante que les deux autres (ce qui est, encore une fois, attendu). La version `OpenMP for` (v5 en rouge) est meilleure que la version `OpenMP task` (v7 en vert). Les raisons sont sans doute les même que pour les versions tuilées.

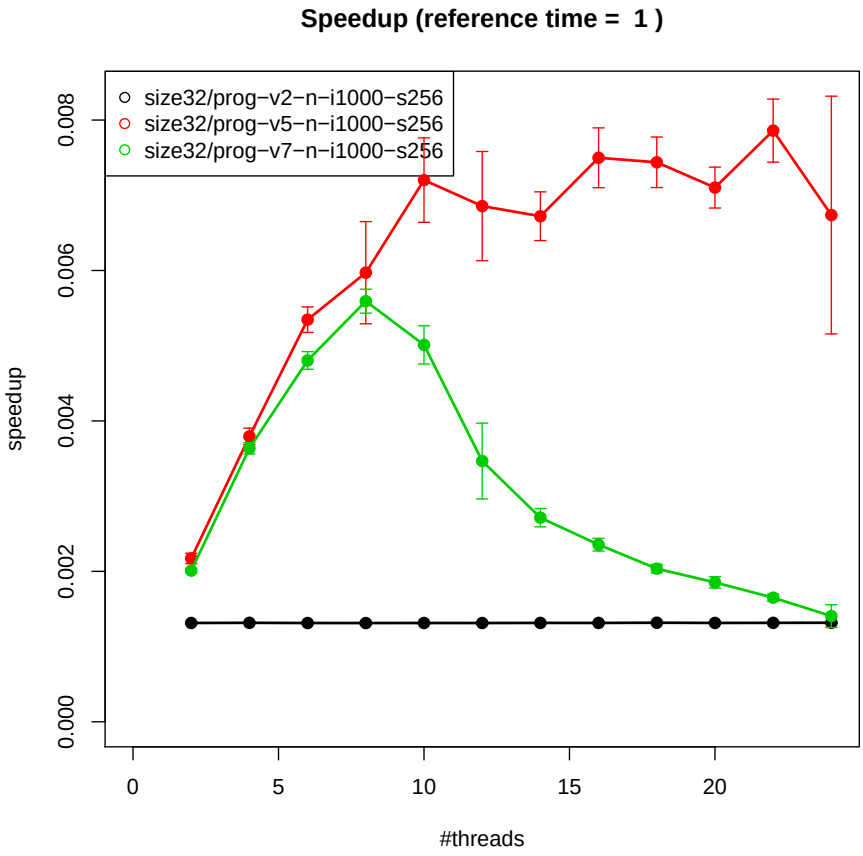

FIGURE 8 – Versions tuilées optimisées sur une image de taille 256 et de tuiles 32

La courbe 9 compare les performances des version tuilées optimisées `OpenMP for` et `OpenMP task` sur une configuration `guns` de taille 1024 avec des tuiles de taille 64. Contrairement à précédemment, la version `OpenMP task` (v7 en rouge) est meilleure que la version `OpenMP for` (v5 en noir). En effet, avec des tuiles plus grandes, le temps de création des tâches est moins long et est compensé par la rapidité d'éxécution des tâches, d'autant plus que les tuiles sont plus grandes donc proportionnellement moins nombreuses.

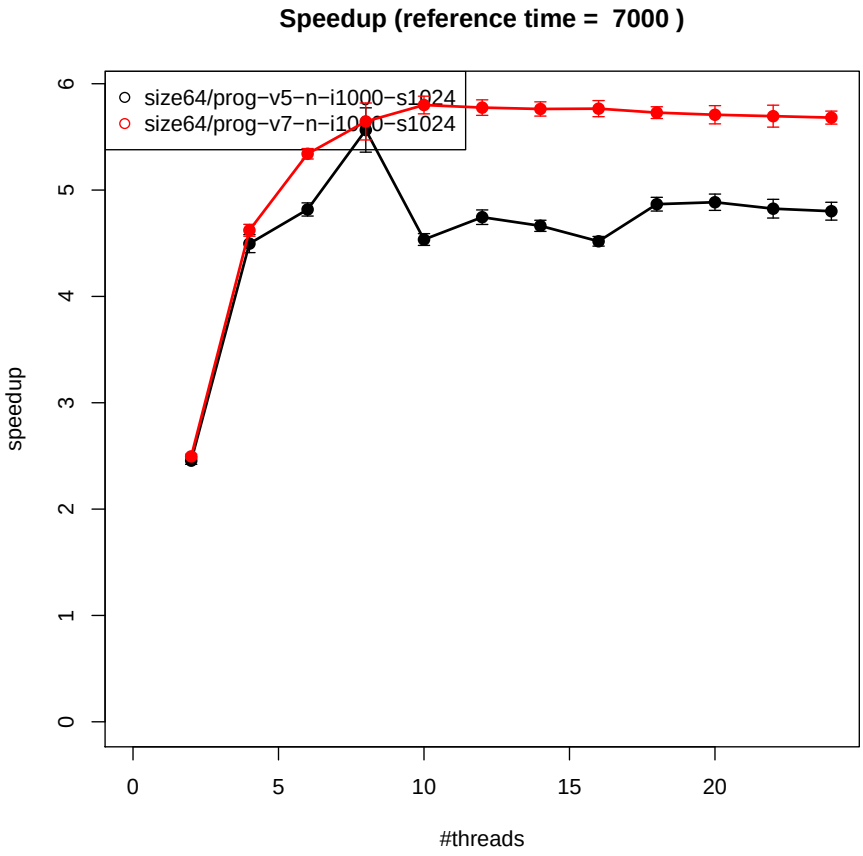

FIGURE 9 – Versions tuilées optimisées sur une image de taille 1024 et de tuiles 64

La courbe 10 compare les performances de la version `OpenMP` tuilée optimisée sur une configuration `guns` de taille 1024 avec des tuiles de taille 32 (en noir) et 64 (en rouge). Nous pouvons constater que de plus grandes tuiles impliquent de meilleures performances à partir d'un certain nombre de threads (ici, la différence se fait à partir de 6 threads).

La différence se montre lorsque le temps perdu à créer les tâches est plus facilement amorti par le grand nombre de threads qui doit les exécuter. L'avantage revient à la courbe rouge car le nombre de tuiles est plus faible (les tuiles étant plus grandes) : peu de tuiles et beaucoup de threads pour les calculer.

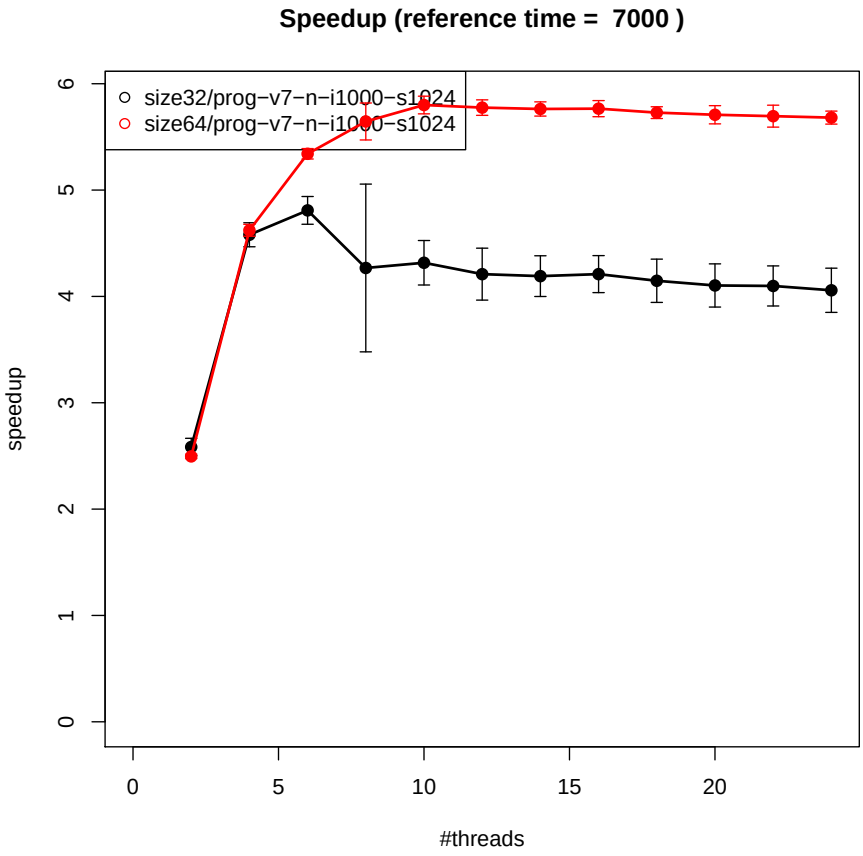

FIGURE 10 – Versions tuilées optimisées `OpenMP task` en changeant la taille des tuiles

2.8 Comparaison des versions `OpenMP` avec `OpenCL`

Sur des images partant d'une configuration `guns` de taille 1024 et dont les tuiles sont de taille 32, la version `OpenCL` optimisée est environ 100 fois plus rapide que les versions tuilées optimisées `OpenMP for` et `OpenMP task`. La figure 11 montre cette accélération : la courbe verte correspond en effet à `OpenCL`. Notons que les courbes `OpenMP` semblent aplaties, car la différence de performances avec `OpenCL` est notable.

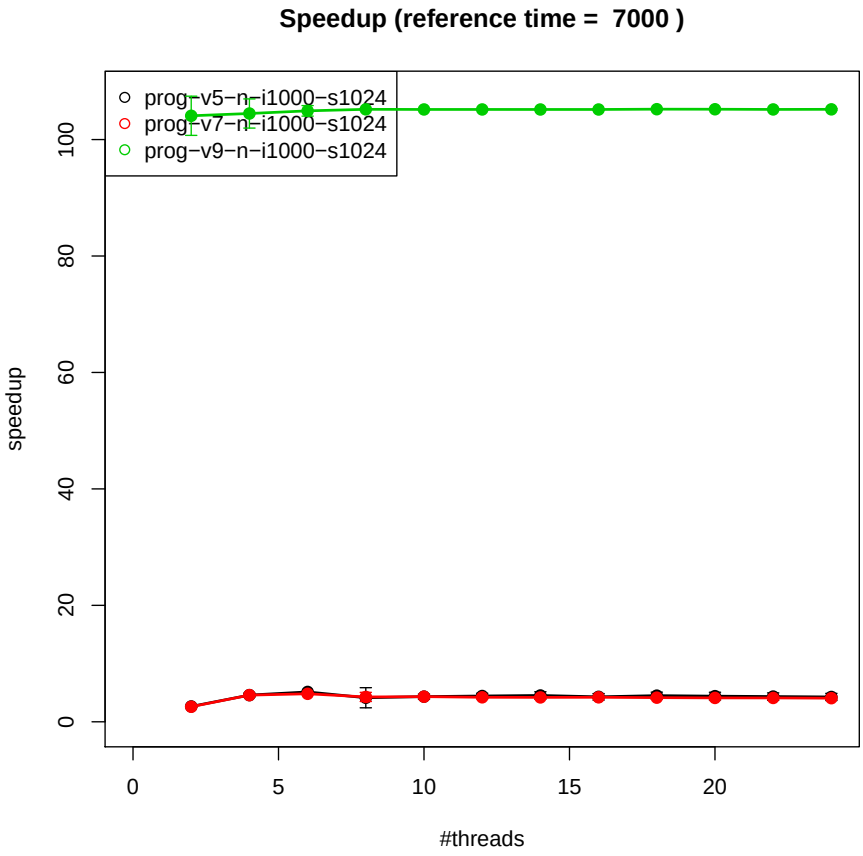

FIGURE 11 – Versions tuilées optimisées et version `OpenCL` optimisée

2.9 Comparaison de l'exécution sur un jeu aléatoire avec une version avec lanceurs

Les courbes noire et verte de la figure 12 correspondent aux versions `OpenMP for` optimisée et `OpenMP task` optimisée utilisant une image de base avec lanceurs, tandis que les courbes rouge et bleue correspondent à leurs homologues sur image de base aléatoire. La taille des images est de 1024, les tuiles sont de taille 32.

Il est clair que l'image aléatoire, dans laquelle les cellules sont plus uniformément réparties, présente moins d'aléas dans les performances mesurées qu'une configuration `guns`. En effet, les images aléatoires ne possèdent par exemple pas (ou peu) de zones stables et permettent moins de mettre à profit

les versions tuilées optimisées. En revanche, les images moins aléatoires comme celles avec lanceurs présentent des performances assez dépendantes de la configuration de base du jeu.

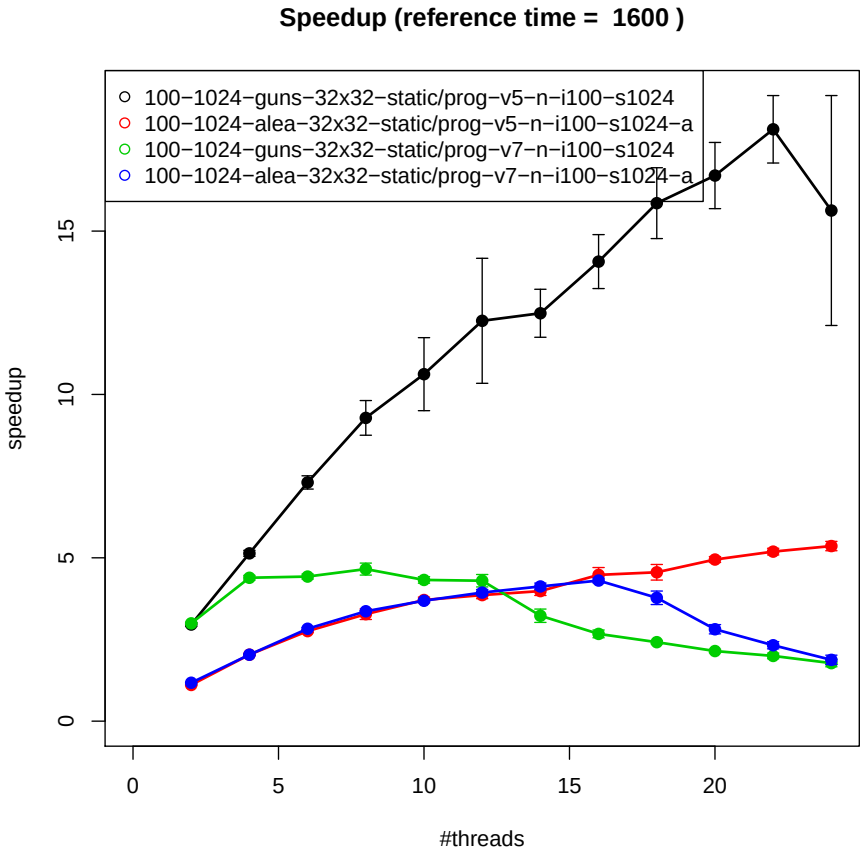

FIGURE 12 – Comparaison de l'exécution sur un jeu aléatoire (courbes rouge et bleue) avec une version avec lanceurs (courbes noire et verte)

2.10 Comparaison des ordonnancements statique et dynamique

La figure 13 présente la différence de performances entre une version tuilée optimisée utilisant un `for collapse(2)` et un ordonnancement statique (en rouge) et la meme version utilisant un ordonnancement dynamique (en noir). L'image de base utilisée comporte des lanceurs et est de taille 1024, les tuiles sont de taille 64.

La version dynamique est initialement moins performante que la version statique, mais la tendance s'inverse dès environ 8 threads : à partir de là, la version dynamique est nettement meilleure. En effet, la version utilisée étant une tuilée optimisée, il peut arriver que des tuiles ne soient pas recalculées. Avec un ordonnancement statique, des threads peuvent alors se retrouver désoeuvrés. En revanche, avec l'ordonnancement dynamique, tous les threads auront du travail tant que l'image complète n'aura pas été parcourue. C'est ainsi que les performances sont améliorées.

Il se peut que les performances soient moins bonnes en dessous de 8 threads, car les threads (en sous-nombre) doivent chercher du travail un grand nombre de fois alors qu'un ordonnancement statique les fixe sur ce qu'ils ont à faire dès le départ.

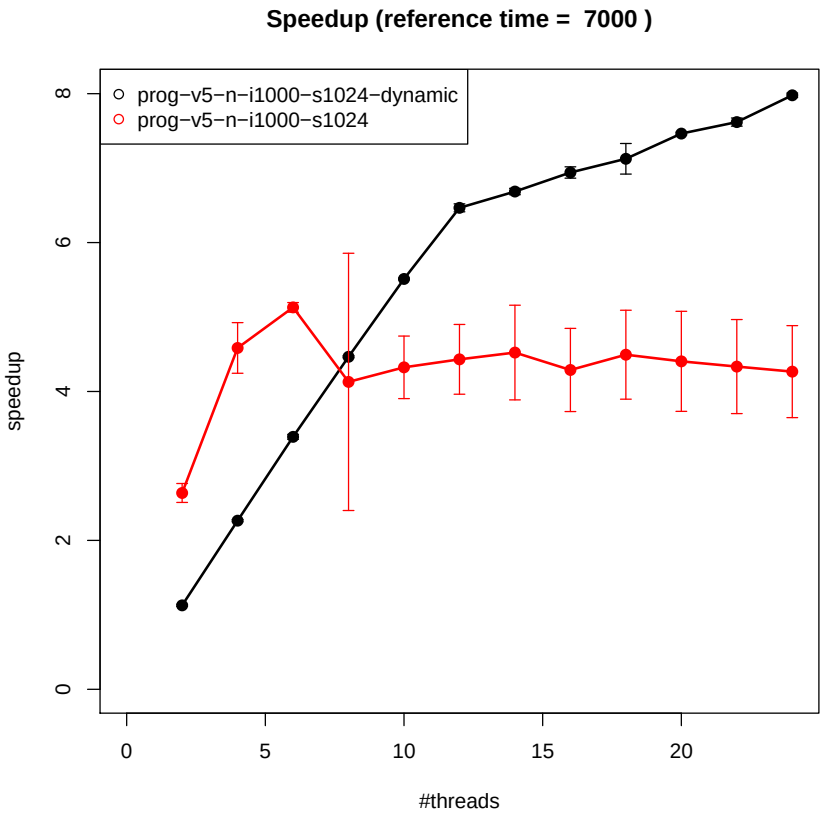

FIGURE 13 – Version tuilée optimisée `OpenMP for` avec ordonnancement statique (rouge) ou dynamique (noir)