Date de rendu : **1 Juin 2018**

Programmation multicoeur et GPU

Jeu de la vie

2ème année Informatique

ENSEIRB-MATMECA - Informatique

Jean-Loup BEAUSSART

Vincent RUELLO

Table des matières

Introduction

L'objet d'étude de ce projet est le jeu de la vie imaginé par John Horton Conway en 1970.

L'objectif est de concevoir et implémenter différents algorithmes permettant de simuler chaque itération du jeu puis de comparer leurs performances.

Dans ce cadre, des versions séquentielles, utilisant Open MP (CPU) ou utilisant Open CL (GPU) ont été mises en place afin d'évaluer leur efficacité.

1 Versions séquentielles

Nous avons écrit 3 versions séquentielles :

- `seq` : Version faisant une boucle sur tous les pixels du jeu pour les mettre à jour à chaque itération.
- `seq_tile` : Version faisant une boucle sur toutes les tuiles puis pour chaque tuile sur tous ses pixels pour les mettre à jour à chaque itération.
- `seq_tile_opt` : Cette version maintient 2 matrices de booléens (1 booléen par tuile) permettant de ne recalculer que les pixels des tuiles qui en ont besoin. Ainsi, on boucle sur toutes les tuiles et pour chaque tuile on regarde s'il est nécessaire de la recalculer, et si besoin on calcule tous ses pixels. Une tuile a besoin d'être recalculée si l'un de ses voisins a été modifié lors de l'itération précédente.

Comparons les performances de ces trois versions en partant du motif "guns" pour 1000 itérations :

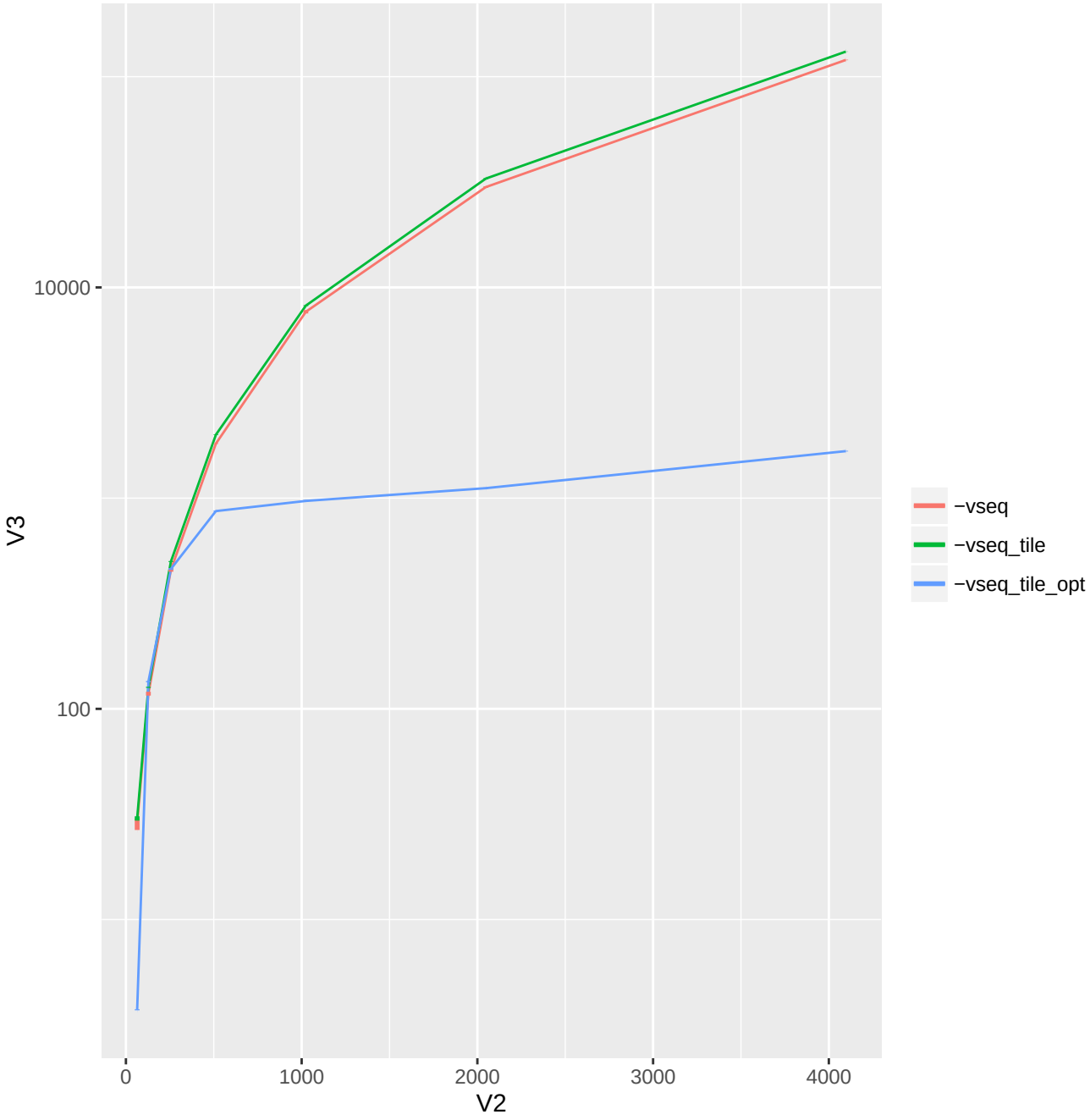

FIGURE 1 – Représentation en échelle logarithmique du temps de calcul en fonction de la taille du monde

Les versions `seq` et `seq_tile` montrent des performances similaires. En effet, dans les deux cas, on calcule tous les pixels de chaque tuile. Seule la syntaxe est différente, syntaxe qui peut d'ailleurs être écrasée par le compilateur.

La version `seq_tile_opt` fournit de meilleurs performances. Cette version effectue pour chaque tuile un calcul supplémentaire, mais permet d'économiser le calcul de

chaque pixel d'une tuile dans certains cas. Avec l'initialisation en "guns", beaucoup de tuiles ne nécessitent pas d'être calculées à chaque itération, ce qui rend cette optimisation valable.

2 Versions Open MP

2.1 Versions sans tâches

Nous avons adapté les 3 algorithmes séquentiels précédents en Open MP à l'aide de `#pragma omp parallel for`.

Pour la version simple, nous avons parallélisé le parcours de tous les pixels avec `#pragma omp parallel for collapse(2)`.

Pour les version avec tuiles, nous avons parallélisé le parcours de toutes les tuiles avec `#pragma omp parallel for collapse(2)`.

Dans chacun de ces algorithmes, nous avons observé que le nombre optimal de threads lancés correspond au nombre de coeurs de la machine hôte.

Voici un comparatif de chacun de ces algorithmes, utilisant un ordonnancement `static` pour 1000 itérations :

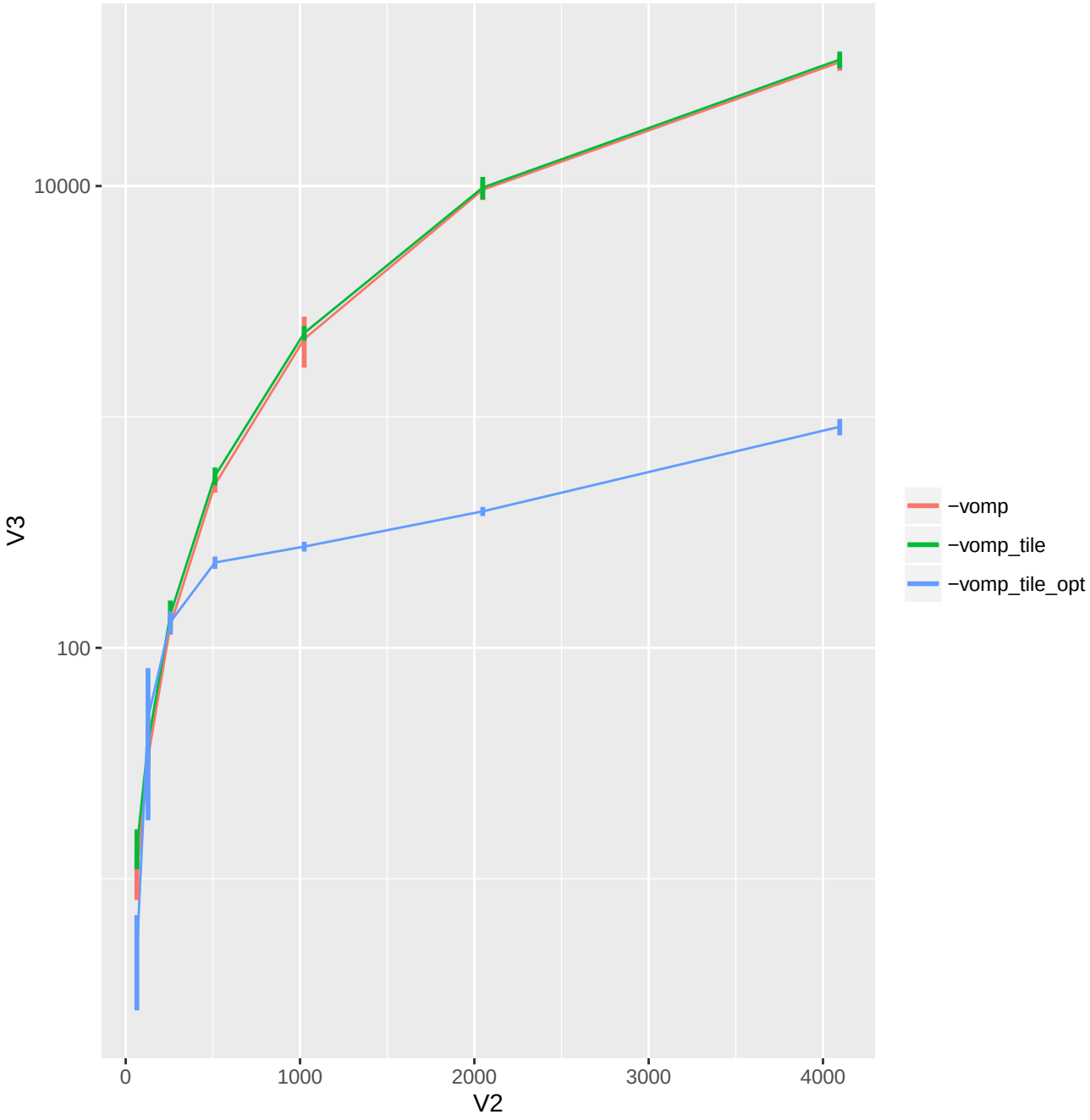

FIGURE 2 – Représentation en échelle logarithmique du temps de calcul en fonction de la taille du monde

2.1.1 Stratégie d'ordonnancement

Nous avons comparé dans les versions `omp`, `omp_tile` et `omp_tile_opt` l'utilisation d'un ordonnancement `static` et `dynamic`. Dans les deux cas, les options par défaut ont été utilisées (en particulier, pour `dynamic`, chaque thread ne récupère qu'un élément

à traiter).

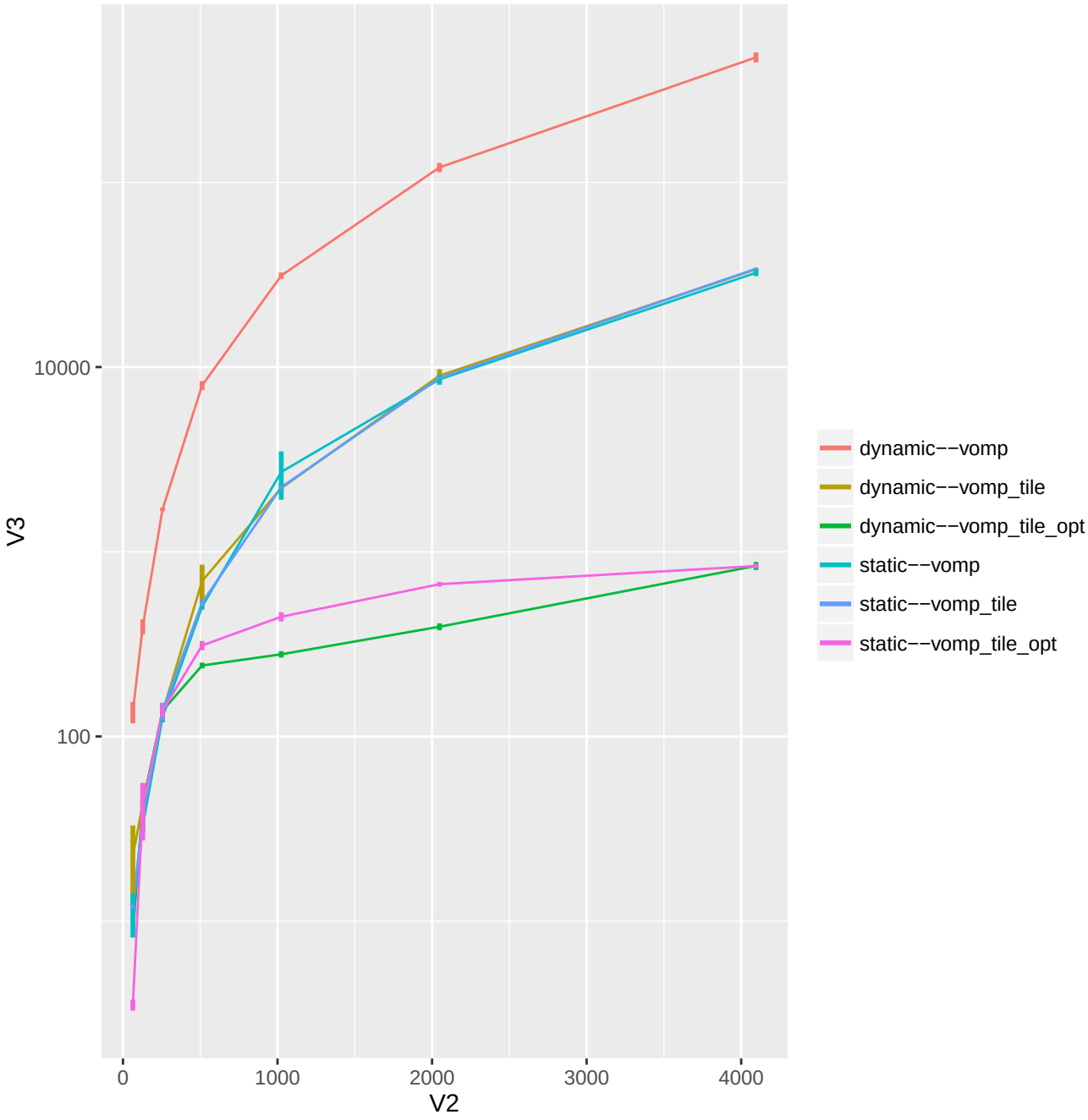

FIGURE 3 – Représentation en échelle logarithmique du temps de calcul en fonction de la taille du monde, toujours pour 1000 itérations

On remarque que dans le cas de `omp_seq`, il est nettement plus performant d'opter pour un ordonnancement `static`. Cela peut s'expliquer par le fait que chaque thread calcule un pixel, et donc tous les threads font la même quantité de calcul, qui est relativement faible. Ainsi, il n'y a pas de threads beaucoup plus rapide que les autres,

et le coût de l'ordonnancement est très lourd par rapport au calcul du thread. En effet, il y a $\texttt{DIM}^2$ pixels à traiter, donc il y a le même nombre d'ordonnancements. La version `dynamic` est donc inefficace.

Dans le cas de `omp_tile`, les deux stratégies semblent fournir des performances similaires. Cette fois, chaque thread calcule une tuile et tous ses pixels. Le calcul du thread est donc plus important et le coût de l'ordonnancement devient moins visible dans les performances. En effet, il n'y a que $\left(\frac{\texttt{DIM}}{\texttt{TILE}}\right)^2$ tuiles à traiter, donc le même nombre d'ordonnancement. Par contre, tous les threads ont la même charge de calcul, donc il n'y a pas de thread qui termine beaucoup plus vite que les autres. Ainsi, les deux stratégies sont équivalentes.

Dans le cas de `omp_tile_opt`, l'ordonnancement `dynamic` est plus efficace. Dans cette situation, chaque thread calcule une tuile, mais les tuiles ne demandent pas le même temps de calcul. En effet, les tuiles dites "à recalculer" nécessitent le calcul de la couleur de tous leurs pixels, alors que les autres n'en ont pas besoin. Ainsi, le traitement de certaines tuiles termine beaucoup plus rapidement que pour d'autres, et il est donc pertinent d'opter pour un ordonnancement `dynamic`.

Le pire des cas est d'utiliser la version `omp` avec un ordonnancement dynamique. Ensuite, les versions `omp_tile` (peu importe l'ordonnancement) et `omp` avec ordonnancement statique sont équivalentes. Enfin, les versions `omp_tile_opt` sont proches.

2.2 Versions avec tâches

Deux versions utilisant le paradigme des tâches ont été mises en place :

— La version `omp_tile_task` consiste en la création d'une tâche par tuile. Cette tâche calcule la couleur de tous les pixels de la tuile. Un pool de threads est lancé avec l'instruction `#pragma omp parallel`, puis un thread unique génère les tâches exécutées par les autres threads.

— La version `omp_tile_task_opt` consiste en la création d'une tâche par tuile. Cette fois, la tâche ne calcule la couleur des pixels de la tuile qu'à condition que la tuile ait besoin d'être recalculée (en utilisant des matrices annexes).

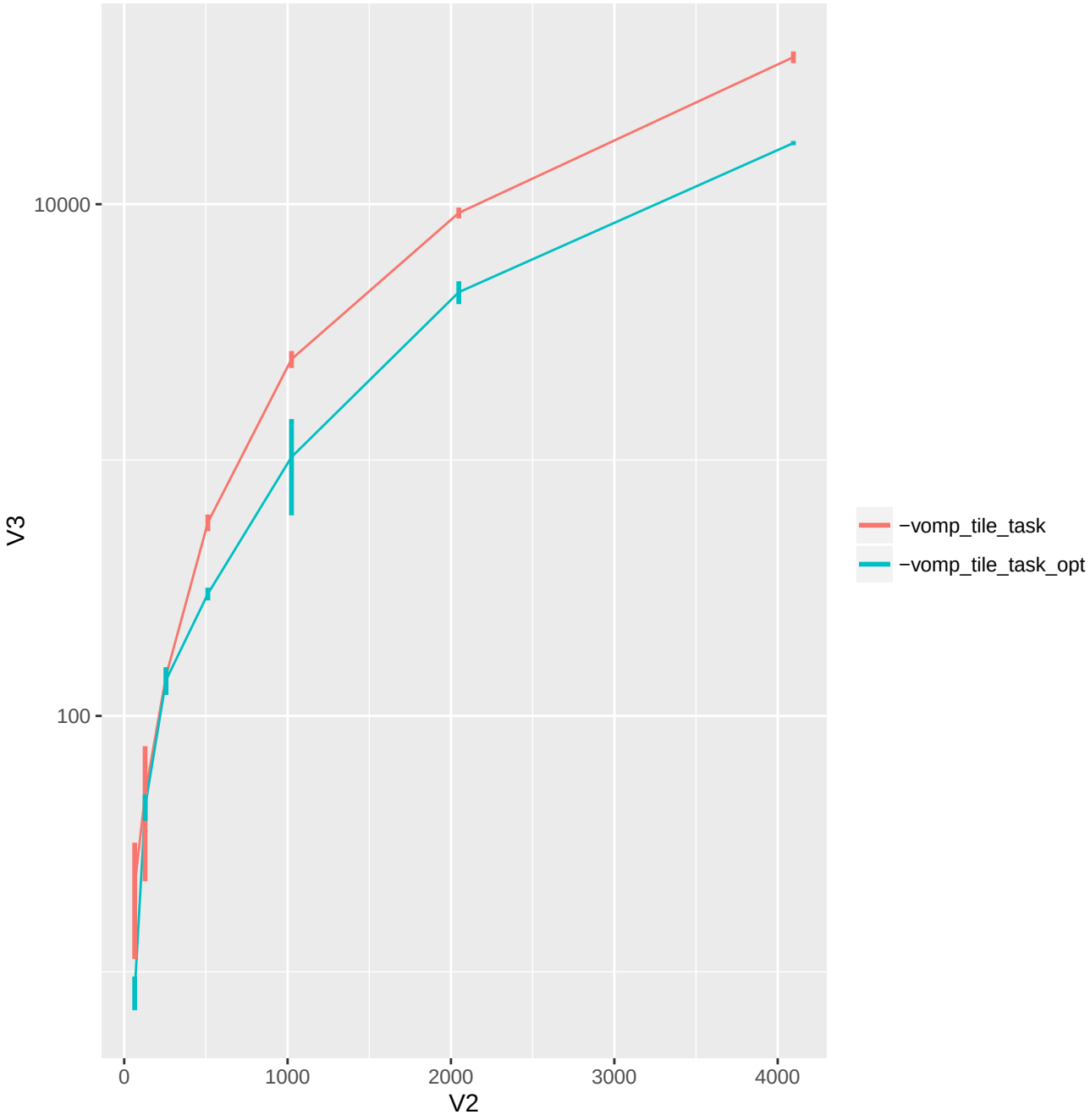

FIGURE 4 – Représentation en échelle logarithmique du temps de calcul en fonction de la taille du monde, pour 1000 itérations

Les performances de ces versions sont similaires, et relativement médiocres (à peine mieux que la version séquentielle basique).

3 Versions Open CL

Dans la version Open CL, chaque pixel est assigné à un thread. Nous avons implémenté 2 kernels : `vie` et `vie_opt`.

Le kernel `vie` calcule simplement la nouvelle couleur du pixel correspondant au thread l'exécutant.

Le kernel `vie_opt` utilise en plus deux tableaux en arguments du kernel qui permettent de suivre les modifications des tuiles. Lors de l'exécution, le premier pixel de chaque tuile vérifie s'il est nécessaire de recalculer la tuile. Si c'est le cas, chaque thread gérant un pixel de la tuile va le recalculer. Sinon, les autres threads ne feront rien.

Comparons les performances de ces deux versions pour 1000 itérations :

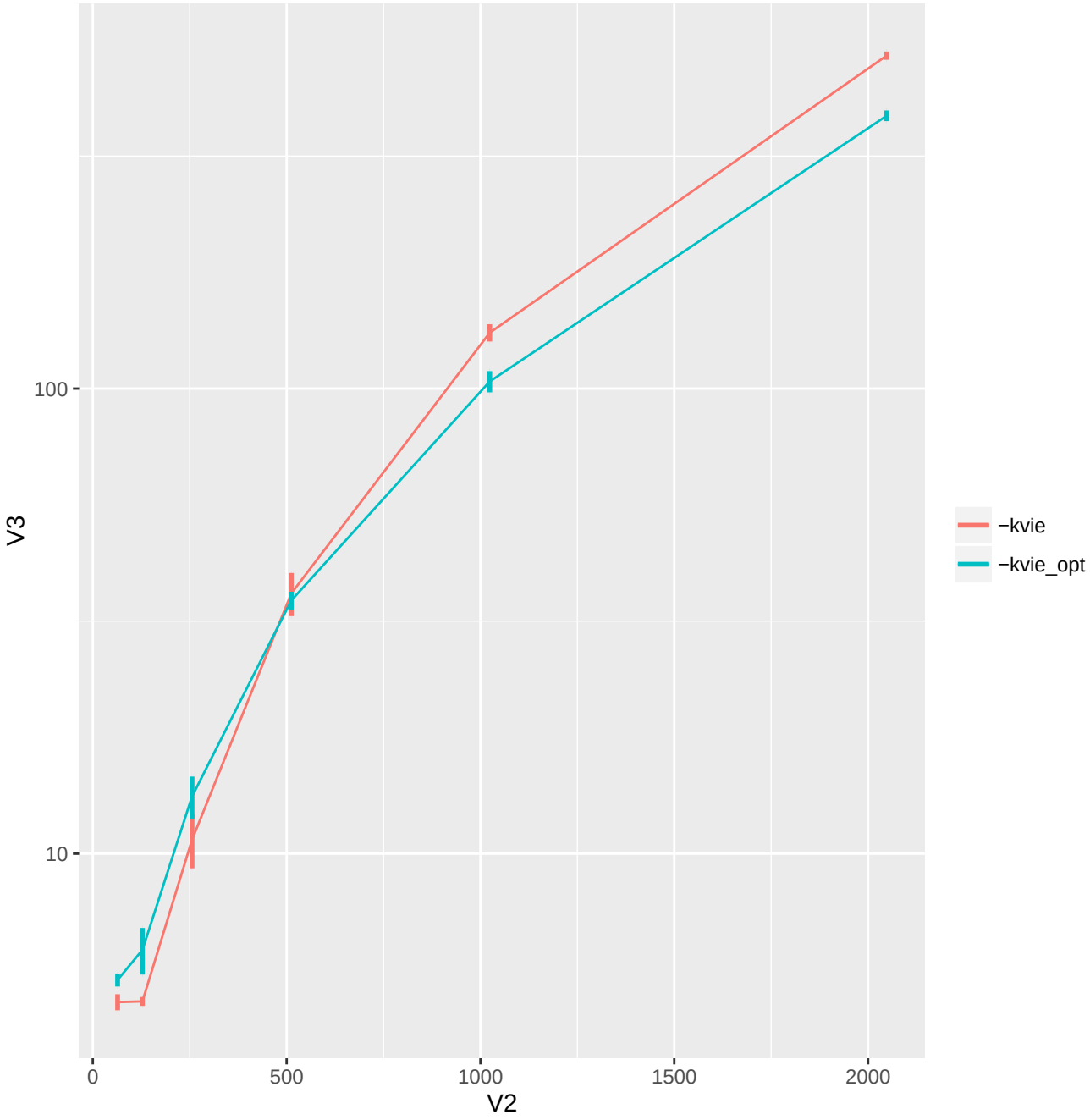

FIGURE 5 – Représentation en échelle logarithmique du temps de calcul en fonction de la taille du monde

On observe que les performances de ces deux stratégies sont relativement similaires, bien qu'on note une amélioration avec `vie_opt` lorsque la taille devient grande.

La similarité entre les deux peut s'expliquer par l'architecture d'un accélérateur. Il contient en effet énormément de processeurs et la création d'un thread ne coûte rien. Ainsi, lancer `nombre de pixels` thread n'est pas choquant et ne détruit pas les per-

formances comme sur CPU. La différence entre les deux se fait donc uniquement sur le fait que `vie_opt` réduit (un peu) le nombre de calculs à effectuer.

Cependant, pour une taille de 4096, le nombre de threads à générer pour `vie` est supérieur à 16 millions, ce qui peut probablement saturer le GPU et expliquer ainsi la léger avantage possédé par `vie_opt`.

Ces versions sont dans les deux cas bien meilleures que celles utilisant Open MP.

Comparaison des différentes méthodes

Pour terminer, nous avons comparé toutes les méthodes programmées sur 1000 itérations de la configuration "guns". Il apparaît que la méthode utilisant OpenCL est beaucoup plus performante, suivie de la version avec tuiles optimisée et OpenMP, suivie de la version avec tuiles optimisée non parallèle. Les plus lentes étant les versions séquentielle et séquentielle avec tuiles.

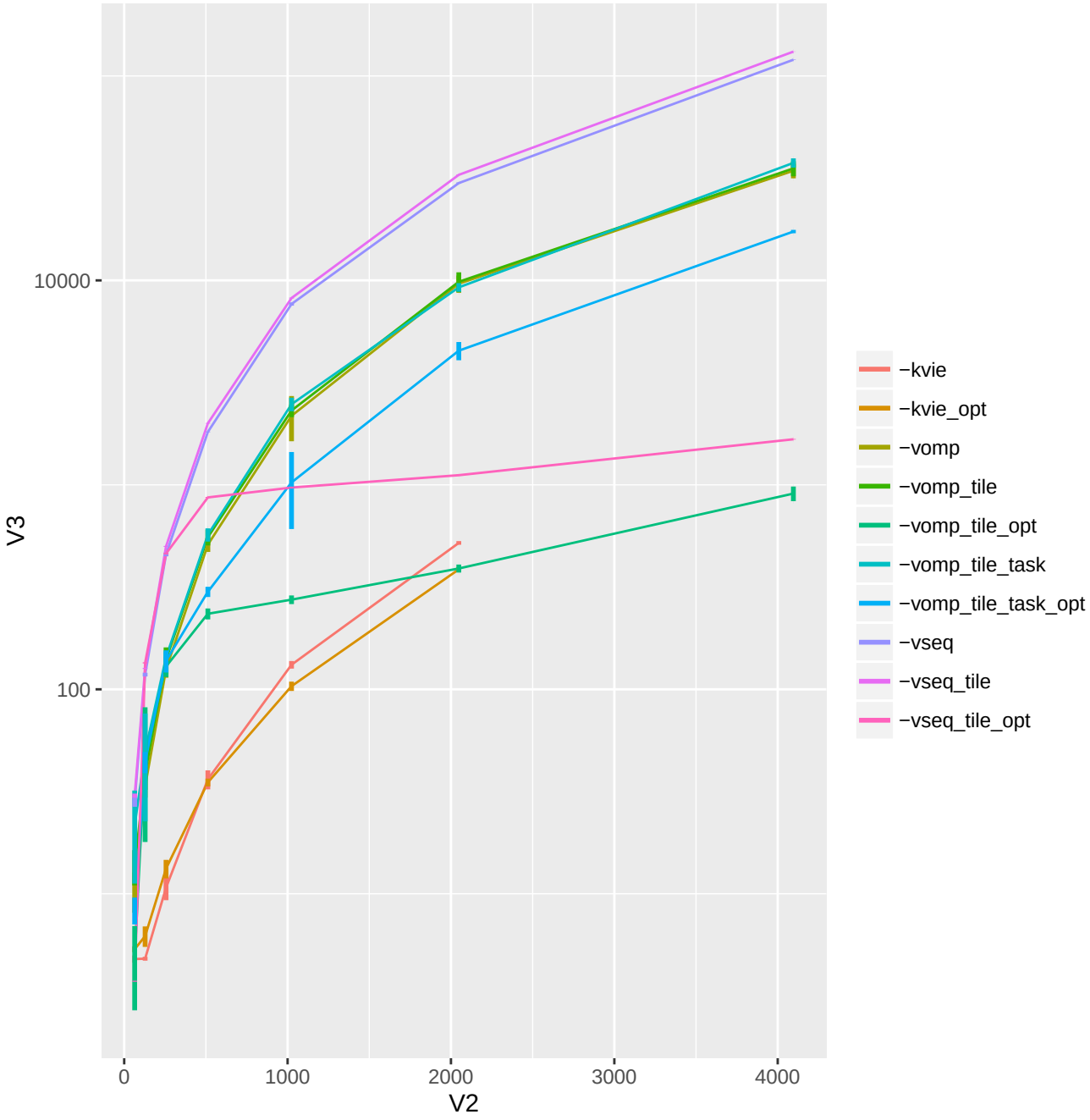

FIGURE 6 – Représentation en échelle logarithmique du temps de calcul en fonction de la taille du monde

Conclusion

Nous avons étudié dans ce rapport l'évolution des performances liées à l'emploi d'algorithmes séquentiels, utilisant Open MP ou Open CL. On observe que dans notre

cas d'utilisation, les algorithmes séquentiels sont moins performants que ceux utilisant Open MP, eux même moins efficaces que ceux utilisant Open CL.

Nous avons travaillé sur un algorithme de tuile optimisée avec toutes les versions. La plupart du temps, il fournissait des performances légèrement meilleures dans le cas d'initialisation avec un motif de type "guns". Dans le cas d'une initialisation aléatoire, il n'y a quasiment jamais de tuiles n'ayant pas besoin d'être recalculées, et donc toutes les tuiles doivent être recalculées à chaque itération (on perd donc toute l'optimisation).