ENSEIRB-MATMECA

Département informatique

Projet de programmation multicoeur et GPU

Jeu de la vie

Auteurs :
Lucas Barros de Assis
Vincent Bridonneau

29 mai 2019

Table des matières

1 Introduction

Ce document est le rapport du projet de *Programmation Multicoeur et GPU* qui consiste à simuler le jeu de la vie avec différentes implémentations. À la fin, nous ferons une analyse des résultats pour chaque algorithme et pour quelques motifs initiaux à la configuration.

Toutes les implémentations et les résultats des tests sont disponibles via le lien https ://github.com/lbassis/jeu_vie.

2 Division de tâches

Les implémentations de base ont été faites en groupe. Après, l'élève Lucas a fait les implémentations *OpenCL* et ses analyses et Vincent les implémentations *OpenMP* et ses analyses.

3 Versions séquentielles

Trois différentes versions séquentielles ont été implémentées. Une première, basée sur celle donnée, en réduisant le nombre de sauts conditionnels, une deuxième tuilée et une troisième tuilée mais sans regarder les tuiles qui n'ont pas changé, à la dernière itération.

3.1 Version de base optimisée

L'idée pour optimiser la version de base est de diminuer le nombre des sauts conditionnels. Nous avons trouvé une façon de le faire en enlevant le test *if (i != y || j != x)* au comptage des voisins. L'idée est de sommer, pour toute cellule c, toutes ses cellules voisines, y compris elle même (pour ne plus avoir le test évoqué précédemment). Ensuite, pour savoir si la cellule sera en vie à l'itération suivante, nous calculons $\mathbf{t = (((n == 3) \mid ((tmp != 0) \,\&\, (n == 4))))}$, où tmp est la valeur de c non encore mise à jour. cette valeur vaut 1 si et seulement si c est en vie et a entre 3 et 4 voisins en vie ou si c est morte et en a exactement 3 voisins en vie. Nous multiplions alors t par une valeur donnant la couleur de la cellule. Elle sera noire si $t = 0$ et jaune si $t = 1$.

Nous avons eu les résultats suivants (avec 500 itérations) :

taille (cellule/ligne)	512	2048
basique (ms)	857.07	13758.83
optimisé (ms)	933.17	15003.09

TABLE 1 – Résultat séquentiel optimisé

On constate que le temps est en faveur de la version de base ce qui peut s'expliquer par la plus grande importance apportée à l'optimisation du saut conditionnel par le compilateur (en l'occurrence gcc 6.3.0) par rapport aux opérations binaires bit à bit.

3.2 Version tuilée

Pour la version tuilée, on va diviser l'image en tuiles de taille *DIM/GRAIN* et appeler la fonction *traiter_tuile*, disponible dès la version de base, pour traiter à chaque fois une tuile différente, en parcourant toute l'image.

3.3 Version tuilée optimisée

Cette version ne calcule pas les valeurs des tuiles qui n'ont pas été modifiées à la dernière itération et dont les voisines n'ont pas été modifiées non plus.

Pour ce faire, il faut d'abord avoir une matrice de vérification qui stocke pour chaque tuile si elle a été modifiée à la dernière itération. Une deuxième matrice de mise à jour est utilisé pour garder les mêmes informations sur l'itération actuel. Pour ne pas avoir de problèmes avec les bords, nous avons décidé de faire deux matrices avec une taille *(GRAIN+2)*(GRAIN+2)*. Les deux matrices sont initialisées avec la valeur *1* à chaque position.

À chaque itération, pour chaque tuile, on vérifie si une voisine a été modifiée. Si il y en a au moins une, l'algorithme calcule sa nouvelle valeur comme pour les versions précédentes et signe la tuile comme modifiée. Sinon, on ne fait pas de calcul pour cette tuile.

À la fin de chaque itération, la matrice actuel est copié vers la matrice de vérification et la nouvelle matrice de mise à jour obtient toutes ses positions misent à *0* pour indiquer que, pour l'instant, aucune tuile n'a été modifiée.

Comme les matrices ont leur première et la dernière lignes et colonnes toujours à *0*, nous avons implémenté des fonctions pour faciliter la recherche et la mise à jour des matrices.

```
unsigned has_tile_changed(int i, int j) {
  i++;
  j++;
  return last_modified[j+(GRAIN+2)*i];
}

void tile_changed(int i, int j, unsigned value) {
  i++;
  j++;
  cur_modified[j+(GRAIN+2)*i] = value;
}
```

FIGURE 1 – Fonctions d'accès aux matrices

4 Versions intermédiaires

4.1 Version OpenMP for

Basique

Pour cette partie, nous avons réutilisé le code de la version séquentielle basique en rajoutant une directive pré-processeur omp pour paralléliser le traitement de chacune des cellules :

```
#pragma omp parallel for schedule(static) collapse(2)
for(int i = 0; i < DIM ; ++i)
  for (int j = 0; j < DIM ; ++j) {
    compute_new_state_opti (i, j);
  }
```

Tuilée

L'idée ici est la même que pour la version séquentielle, à ceci près que l'on rajoute une directive pré-processeur omp pour paralléliser le traitement de chacune des tuiles. Le code obtenue est :

```
#pragma omp parallel for collapse(2) schedule(dynamic)
for (unsigned i = 0; i < nbit; ++i) {
  for (int j = 0 ; j < nbit; ++j) {
```

```
    traiter_tuile_omp (GRAIN * i, GRAIN * j, GRAIN * (i +1), GRAIN * (j
  }
}
```

La fonction traiter_tuile_omp s'occupe de mettre à jour chaque cellule de la tuile en appliquant la routine compute_new_state à chaque cellule de la tuile.

Optimisée

Pour cette version, nous avons introduit deux tableaux. Un tableau 'changes' qui pour chaque tuile va indiquer si la tuile a été changé au court de l'itération. Ce tableau est initialisé à 0 au début de chaque itération. Le deuxième tableau est "would_change" qui indique, pour chaque tuile, si elle doit être examiné (i.e. si elle ou une de ses voisines a été modifiées au court de l'itération précédente). Ce tableau est initialisé à une valeur non nulle avant le début de la boucle et à chaque itération, il est mis à jour en fonction de "changes" (`would_change[tuile] = OU_Binaire(changes[cellules voisines]`). La fonction mettant à jour la grille est la suivante :

```
#pragma omp parallel for collapse(2) schedule(dynamic)
for (unsigned i = 0; i < nbit; ++i) {
  for (int j = 0 ; j < nbit; ++j) {
    if (would_change[i * nbit + j])
    changes[i * nbit + j] |= traiter_tuile_omp (GRAIN * i, GRAIN * j, G
  }
}
```

La fonction mettant à jour le tableau "would_change" est :

```
int changed = 0;
int nbit = DIM / GRAIN;

for (int i = 0; i < nbit; ++i) {
    for(int j = 0; j < nbit; ++j) {
      would[i * nbit + j] = 0;
      for (int row = i - 1; row < i + 1; ++row){
        for (int col = j - 1; col < j + 1; ++col)
        if (row >= 0 && row < nbit && col >= 0 && col < nbit) {
        would[i * nbit + j] |= changes[row * nbit + col];
```

```
      }
      }
      if( !changed && changes[i * nbit + j])
        changed = 1;
    }
}
memset(changes, 0, sizeof(int) * nbit * nbit);
return changed;
```

4.2 Version OpenMP task

Tuilée

Pour cette version, nous nous basons sur le code de la version tuilée séquentielle non optimisée. À chaque itération, nous parallélisons une double boucle for (for imbriqué) et au lieu de traiter une tuile, nous créons une tâche. À la fin de la double boucle for, nous lançons toutes les tâches (directives “omp taskwait"). Puis un seul thread s'occupe de mettre à jour la grille. Cela nous donne le code suivant pour une instruction :

```
#pragma omp parallel
    {
#pragma omp for schedule(static) collapse(2)
        for (unsigned i = 0; i < nbit; ++i) {
          for (int j = 0 ; j < nbit; ++j) {
#pragma omp task
            traiter_tuile_omp (GRAIN * i, GRAIN * j, GRAIN * (i +1), GR
          }
        }
#pragma omp taskwait

#pragma omp single
        swap_images ();
```

Optimisée

Pour cette version, nous nous sommes appuyés sur la version séquentielle optimisée. De plus, comme dans la version parallèle “omp for omptimisée",

nous avons créé deux tableaux ‘changes’ et ‘would_change’ dont les fonctions sont les mêmes que dans “omp for optimisée". Pour créer les tâches, nous faisons une double boucle for parallélisée. Nous utilisons la directive “omp task" pour créer les tâches. La tâche créée se chargera de mettre à jour la tuile si “would_change" l’indique. La mise à jour de “would_change" est aussi parallélisé. Pour la mise à jour de la grille sur une itération, on obtient :

```
for (unsigned it = 1; it <= nb_iter; it++) {
#pragma omp parallel
    {
#pragma omp for schedule(static) collapse(2)
    for (unsigned i = 0; i < nbit; ++i) {
        for (int j = 0 ; j < nbit; ++j)
#pragma omp task
        {
            if (would_change[i * nbit + j]) {
                changes[i * nbit + j] |= traiter_tuile_omp (GRAIN * i, G
            } // if
        } // task
    } // for i
#pragma omp taskwait

#pragma omp single
    swap_images ();
    } // omp parallel
update_change(changes, would_change);
} // for it
```

La mise à jour de “would_change" est effectuée par :

```
int nbit = DIM / GRAIN;

#pragma omp parallel
{
#pragma omp for schedule(static) collapse(2)
    for (int i = 0; i < nbit; ++i)
        for(int j = 0; j < nbit; ++j)
#pragma omp task
        {
        would[i * nbit + j] = 0;
```

```
            for (int row = i - 1; row < i + 1; ++row){
                for (int col = j - 1; col < j + 1; ++col)
                if (row >= 0 && row < nbit && col >= 0 && col < nbit) {
                    would[i * nbit + j] |= changes[row * nbit + col];
                }
            }
    }
#pragma omp taskwait
}
memset(changes, 0, sizeof(int) * nbit * nbit);
```

4.3 Version OpenCL

Même si ce n'est pas démandé dans le sujet, nous avons implémenté une première version *OpenCL* sans utiliser la mémoire local des *GPU* pour mieux comprendre comment travailler avec les *kernel*.

Dans ce cas-là, chaque coeur traite une cellule différente de cordonnées (*get_global_id(0)*,*get_global_id(1)*) donc tout qu'il faut faire c'est de calculer les voisines pour définir la couleur qui dit si la cellule est vivante ou morte. Après, chaque coeur donne la bonne couleur à sa cellule via la matrice de sortie.

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

  unsigned color, neighboors = 0;

  if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1) {
    for (int i = y - 1; i <= y + 1; i++) {
      for (int j = x - 1; j <= x + 1; j++) {
        if (in[i*DIM+j] != 0)
          neighboors++;
      }
    }
  }

  if (in[y*DIM+x] != 0) {
    if (neighboors != 3 && neighboors != 4) {
      color = 0;
    }
    else {
      color = 0xFFFF00FF;
    }
  }
  else {
    if (neighboors == 3) {
      color = 0xFFFF00FF;
    }
    else {
      color = 0;
    }
  }
  out[y*DIM+x] = color;
}
```

FIGURE 2 – Implémentation global en *OpenCL*

4.4 Version OpenCL tuilée

Pour mieux profiter de l'architecture des *GPU*s, nous pouvons utiliser la mémoire locale pour stocker une tuile complète. De cette façon, les accès pour calculer la quantité de voisines seront largement plus rapides, sauf pour les bordes. Pour traiter les bords, deux options sont possibles : garder la tuile avec deux lignes/colonnes en plus (celle-ci on l'appellera *local*) ou consulter la mémoire globale quand nous sommes sur les bords (celle-là on l'appellera *semi-local*). En gros, soit nous accédons à la mémoire globale au début pour copier toutes les voisines dans la mémoire locale, soit plus tard pour tester les valeurs des bords. Nous avons implémenté les deux techniques pour regarder les différences au niveau de la performance.

Au début de cette version, il faut que chaque coeur copie sa cellule dans la mémoire locale. (et aussi ses voisines, dans le cas local). Une appel à

la fonction *barrier* fait que les *threads* attendent que tous les autres aient terminé leur tâche. Ensuite, il suffit de faire le même algorithme de comptage de voisines qu'avant, sauf que maintenant on consulte la mémoire locale (sauf sur les bords, à l'implémentation semi-local).

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

  __local unsigned tile[TILEY+2][TILEX+2];

  int x_loc = get_local_id(0);
  int y_loc = get_local_id(1);


  unsigned neighboors = 0;
  unsigned color;

  tile [x_loc+1][y_loc+1] = in[y*DIM+x];
  if (x > 0 && x < DIM - 1 && y > 0 && y < DIM - 1) { //si on est sur la borde, on peut prendre tous les voisines
    tile[x_loc][y_loc] = in[(y-1)*DIM+(x-1)];
    tile[x_loc][y_loc+1] = in[y*DIM+(x-1)];
    tile[x_loc][y_loc+2] = in[(y+1)*DIM+(x-1)];

    tile[x_loc+2][y_loc] = in[(y-1)*DIM+(x+1)];
    tile[x_loc+2][y_loc+1] = in[y*DIM+(x+1)];
    tile[x_loc+2][y_loc+2] = in[(y+1)*DIM+(x+1)];

    tile[x_loc+1][y_loc+2] = in[(y+1)*DIM+x];
    tile[x_loc+1][y_loc] = in[(y-1)*DIM+x];
  }

  barrier(CLK_LOCAL_MEM_FENCE);
```

FIGURE 3 – Étape de copie des voisines dans l'implémentation local

5 Version avancée : OpenCL optimisée

Pour faire la version optimisée en *OpenCL* il faut d'abord ajouter des *buffers* lors de l'appel au *kernel* : un pour avoir la matrice des tuiles changées à la dernière itération, un pour l'itération actuelle et un pour la taille du grain (nécessaire pour calculer les positions dans les matrices de modification et vérification). Aussi, au niveau du code *C*, il faut faire que la matrice de modification devienne celle de vérification, et que la matrice de vérification ne contienne que des *0*, pour dire que, pour l'instant, aucune tuile n'a été modifiée.

Au niveau du *kernel*, l'implémentation est tout simplement une version *OpenCL* de la version tuilée optimisée séquentielle. Le calcul est fait de façon globale, parce que l'algorithme qui n'utilise pas la mémoire locale s'est montré plus rapide dans les tests, sauf que les tuiles qui n'ont pas été touchées et dont les voisines aussi n'ont pas été modifiées à la dernière itération ne sont pas calculé.

Finalement, pour bien *debugger* pour être sûr que seulement les bonnes tuiles sont calculés, nous avons implémenté une fonction *print_changed* qui copie le comportement du monitor utilisée pour identifier les *threads* pour les versions *OpenMP*.

```
static void print_changed(__global unsigned *in, __global unsigned *out, unsigned grain) {

  for (int i = 1; i < grain+1; i++) {
    for (int j = 1; j < grain+1; j++) {
      if (out[i*(grain+2)+j] == 1)
        printf("*");
      else
        printf("-");
    }
    printf("\n");
  }
}
```

FIGURE 4 – Fonction de *debuggage* pour la version *OpenCL* optimisée

6 Résultats

Pour chaque implémentation, nous avons testé l'exécution avec des paramètres différents. Les configurations *guns* et *random* ont été utilisées, pour avoir un cas plus ou moins ordonné et un cas chaotique. Les temps ont tous étés obtenus à partir de la moyenne arithmétique de cinq exécutions de 500 itérations chacune (Sauf pour la version OpenMP). La version que nous appelons "séquentielle" est celle de base optimisée, dont le saut conditionnel a été enlevé. Bien que les graphiques montrent des courbes linéaires, ce format ne reflète pas forcement le comportement du programme.

Les ordinateurs utilisés ont une carte *NVIDIA GTX2070* et un processuer *Intel(R) Xeon Gold 5118*.

6.1 Implémentations de base

Les premiers tests analysent le comportement des implémentations en fonction de la taille du monde. Pour ces tests la taille du grain a été fixée à 16.

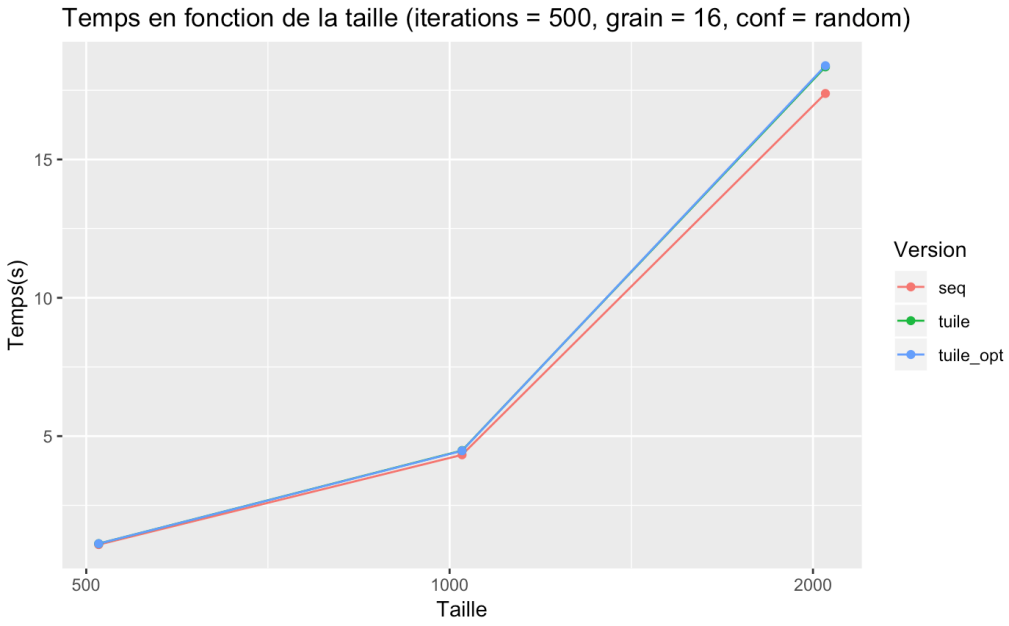

FIGURE 5 – Temps en fonction de la taille avec configuration *random*

Avec la configuration *random*, les temps pour la version tuilée et l'optimisée sont extrêmement proches. Cela arrive grâce à la distribution de points de cette configuration, qui fait qu'il n'y a pas beaucoup de tuiles qui n'ont pas besoin d'être calculées.

Par contre, quand nous observons les temps pour la configuration *guns*, le gain obtenu devient évident. En fait, même pour la taille de 1024 nous avons toujours un temps moins grand avec la version optimisée que pour la taille de 512 avec la version séquentielle. Comme cette configuration a beaucoup d'espaces libres, il est normal que des dimensions plus grande n'affectent pas trop la performance si on calcule seulement le nécessaire.

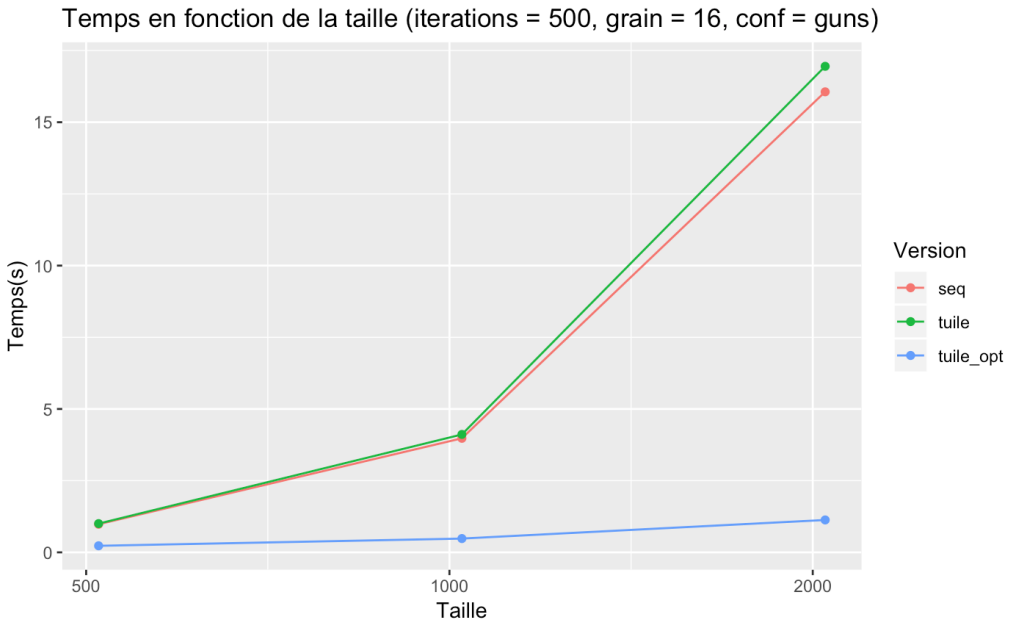

FIGURE 6 – Temps en fonction de la taille avec configuration *guns*

Après, pour vérifier les effets du grain, la taille du monde a été fixée en 512 pour faire des nouveaux tests.

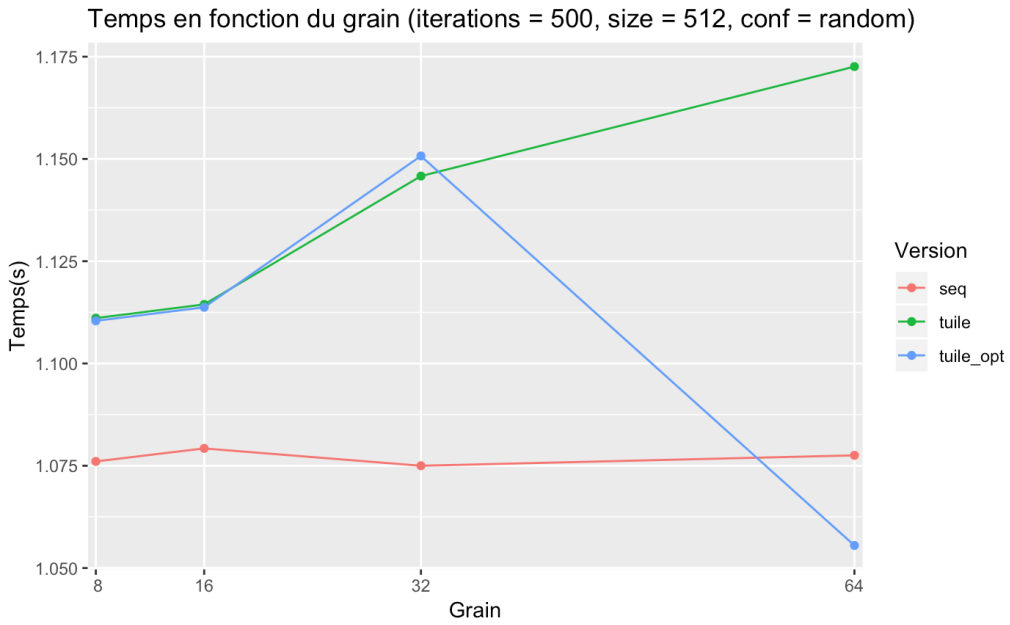

FIGURE 7 – Temps en fonction du grain avec configuration *random*

Pour la configuration *random*, normalement les implémentations tuilées prennent plus de temps. Les tuiles font que les accès à la mémoire ne sont pas pas trop coûteux, et encore, sans utiliser le parallélisme, cette technique ne nous ajoute rien au niveau de la performance. L'exception est quand nous utilisons la version optimisée, où nous commençons à ne pas calculer les tuiles inutiles. Pourtant, comme cette configuration contient des points bien distribués, il faut avoir un grain assez grand pour avoir une accélération considérable. Pourtant, lorsque cette taille de grain est trouvée, nous pouvons avoir de grandes améliorations comme on peut le voir sur la figure 7.

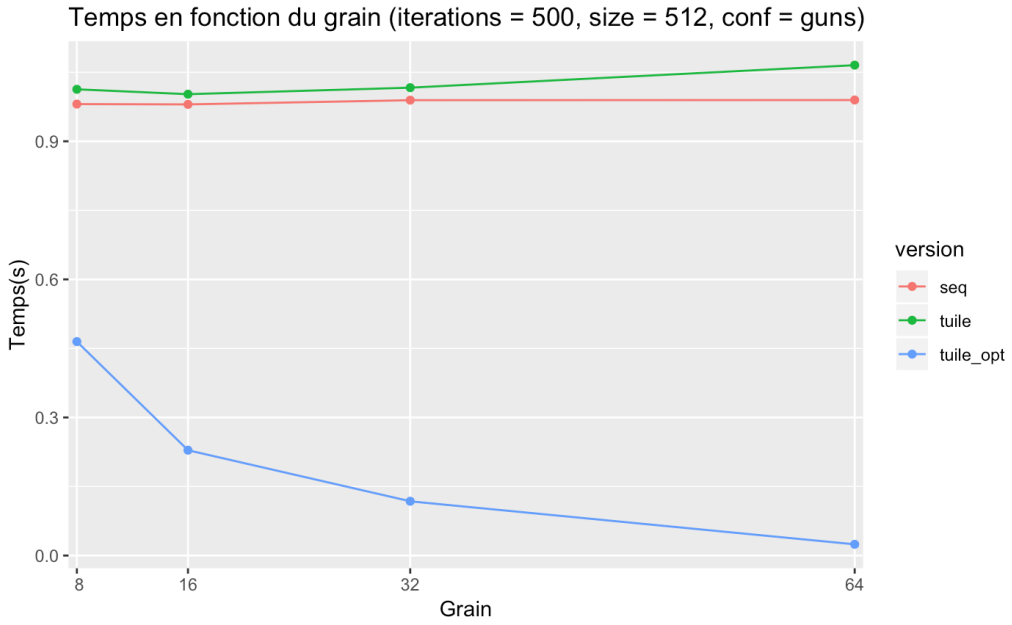

FIGURE 8 – Temps en fonction du grain avec configuration *guns*

Pour la configuration *guns*, nous arrivons à mieux profiter de l'optimisation parce que nous avons des points seulement sur quelques régions de l'image. Ainsi, l'amélioration temporale est présente même avec des grains petits.

6.2 Implémentations *OpenMP*

Voyons à présent les résultats observés pour les versions OpenMP. Le temps de référence pour la version séquentiel est de 164 ms pour $s = 512$ et de 70100 ms pour $s = 4096$. Dans cette partie, on ne s'intéresse qu'au cas guns.

For

Les accélérations obtenus pour la version omp for sont illustré sur la figure 9.

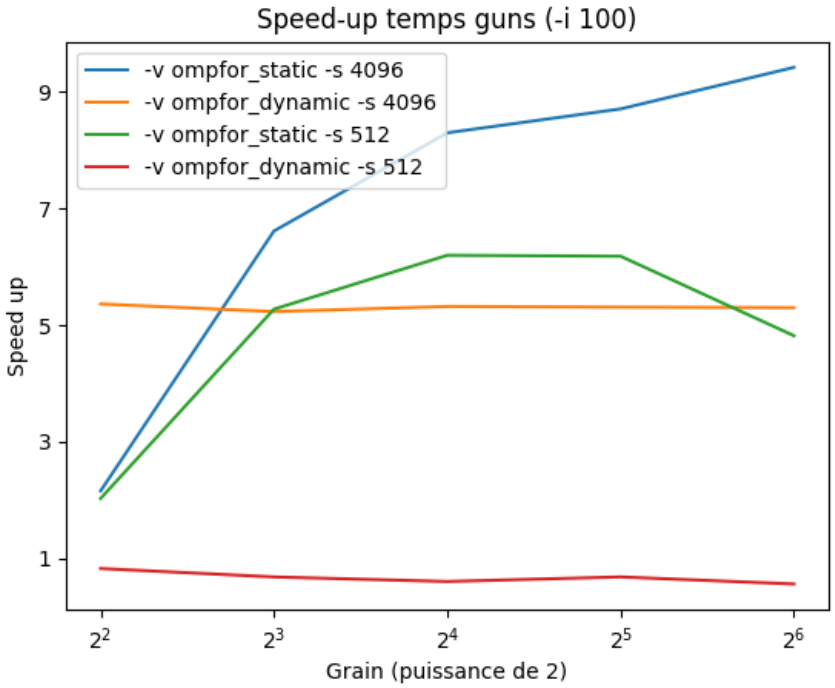

FIGURE 9 – Speed up pour la version OMP for basique

Comme on peut le voir sur la figure 9, les gains en static sont meilleurs quand la taille du grain est entre 8 et 32 (voire très bien quand il vaut 64 et que la taille de la grille est de 4096 cellules de côté). Cela est dû au fait que les tuiles sont proches en mémoire. Il est mauvais cependant quand l'ordonnancement est dynamic pour une petite taille (512), à cause du temps mis par l'ordonnanceur pour allouer une cellule à un thread.

Pour ce qui est de la version tuilé, nous avons la figure 10 pour illustrer les résultats. (Par rapport à la légende, il faut inverser statique et dynamique sur les courbes).

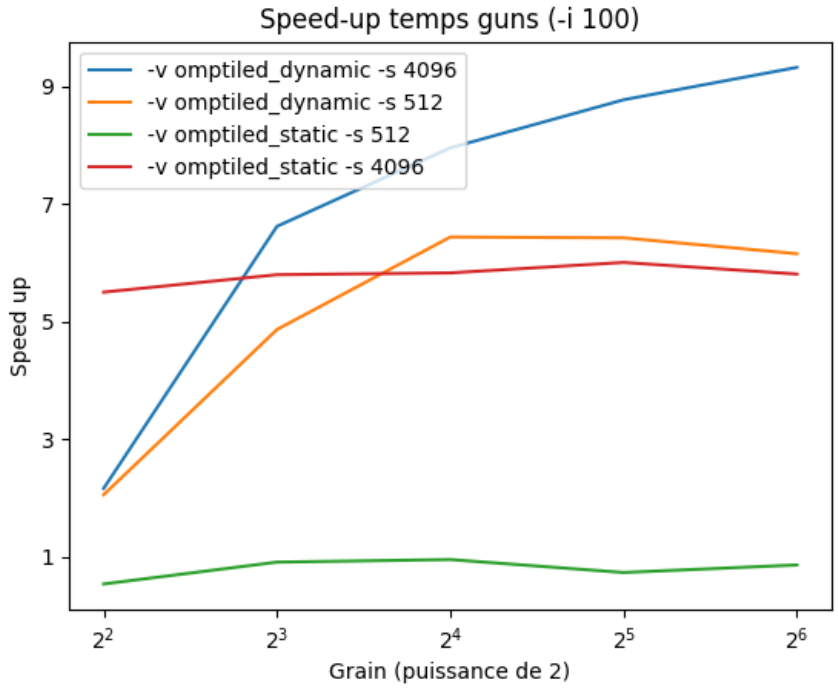

FIGURE 10 – Speed up pour la version OMP for tuilée

Comme on le voit l'accélération est meilleure dans le cas dynamique que dans le cas statique à partir d'une certaine taille de grain (4). Cela n'est pas du à un problème de localité en mémoire car quand la taille des cellules est grande (4096), l'accélération augmente. Cette différence doit s'expliquer par le fait que quand la taille du problème est petite, l'ordonnanceur doit perdre du temps à attribuer une nouvelle tâche à chaque thread, ce temps devient négligeable quand la taille du problème grandi.

On remarque qu'entre la version "omp for" et la version "omp tuilée" les courbes sont semblables et sont relativement proches.

En ce qui concerne la version optimisée, on obtient les courbes suivantes :

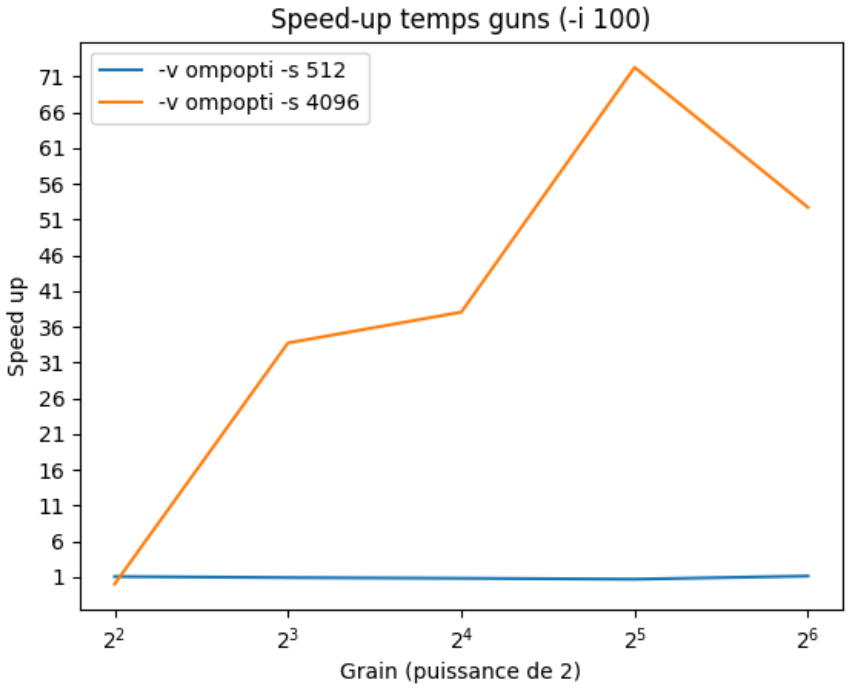

FIGURE 11 – Speed up pour la version omp for tuilée optimisé (statique)

Remarque : un défaut dans le code provoquait une erreur de segmentation quand le grain était de taille 4, du coup, pour ne pas avoir de problème avec les scripts, nous avons mis une très grande valeur pour cette taille de grain pour avoir un point à 1 sur la courbe (uniquement quand la taille du problème $s = 4096$).

On constate ici que l'accélération est très importante et qu'elle dépasse notamment le nombre de threads sur la machine, qui est de 24, quand $s = 4096$. Cela s'explique par le fait que dans la version guns, peut de tuile sont changeantes, au début notamment il n'y a que les coins ce qui fait que sur les 100 premières itérations, il n'y a rien à faire. Cela montre que la version séquentiel de base est mal implémenté car elle ne prend pas en compte les tuiles qui ne vont pas bouger. La tendance semble s'inverser pour grain = 64, la taille de la tuile étant alors très petite, le nombre de d'itération dans la double boucle for imbriqué commence à diminuer, et ainsi certains threads n'ont rien à faire car la condition donné par "would_change" pour la tuile qu'ils ont à traiter est fausse. Il y a donc une mauvaise répartition du

travail. C'est d'autant plus visible quand la taille du problème est de 512, car là l'accélération n'est pas visible, sans doute aussi à cause d'une mauvaise répartition du travail.

Tâches

Pour la version "omp task", les résultats sont les suivants :

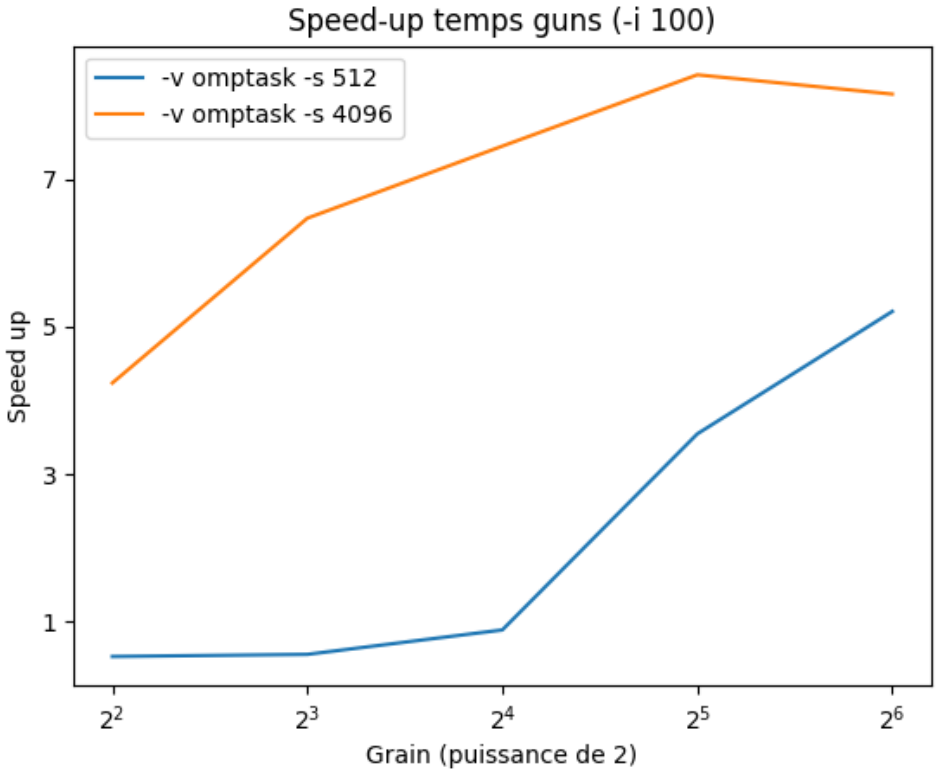

FIGURE 12 – Speed up pour la version omp task tuilée

On constate que quand la taille du problème est petite, l'accélération est moins importante et que quand le grain augmente la vitesse également. Pour le deuxième point (augmentation de la taille du grain implique accélération plus élevé), le nombre d'itération dans la double boucle for imbriqué diminue

et donc le nombre de tâche créé diminue. ainsi les threads on un petit nombre de tâche et demande donc moins de tâches à réaliser.

Regardons maintenant ce qui se passe dans le cas optimisé :

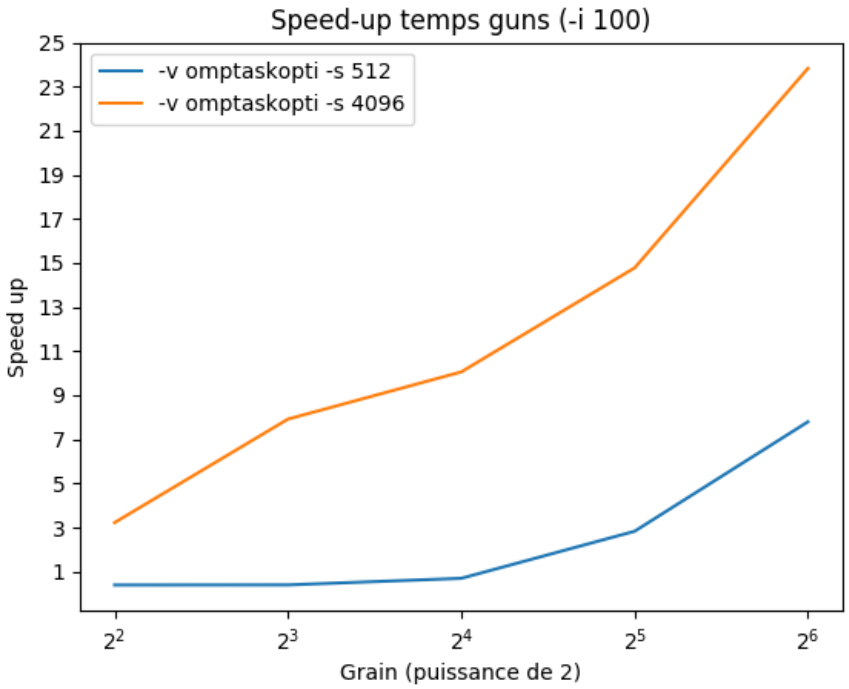

FIGURE 13 – Caption

L'allure des courbes dans le cas du problème petit est la même que ce soit optimisé ou non, seul la valeur du speed up change. L'accélération s'envole quand s est grand, mais elle n'atteint pas la même valeur que pour omp for optimisé. Sans doute pouvons nous gagner du temps en réussissant à enlever les tâches inutiles (ne faire une tâche que si would_change l'indique et non pas faire une tâche qui ne fera qu'un saut conditionnel si would_change est fausse).

Comme on a pu le voir avec ces courbes, les versions optimisés sont nettement meilleurs que les versions non optimisés car elles ne traitent pas des tuiles non nécessaires. De plus il semble que l'ordonnancement statique soit meilleur que le dynamique, ce qui peut surprendre un peu car on a tendance

à ce dire que dans le cas dynamique dès qu'un thread est libre il traite des tuile directement.

6.3 Implémentations *OpenCL*

Au total, quatre implémentations différentes ont été implémentées en *OpenCL*. Ainsi comme pour les versions basiques, nous observons les performances en fonction de la taille et du grain pour les configurations *guns* et *random*.

Pour la configuration *random*, la version optimisée a été la pire des quatre. Nous pensons que, dans cette configuration nous n'avons pas beaucoup des tuiles libres, les calculs extra pour savoir quelles sont les tuiles à modifier prend trop de temps sans apporter une amélioration perceptible.

La version semi-local s'est montré, de façon inattendue, un mauvais choix. Bien qu'elle utilise la mémoire locale, il faut accéder à la mémoire globale pour calculer les voisines sur les bords. Donc, le plus l'image est grande, plus il y a de tuiles sur les bords.

La différence entre les versions globale et locale a été aussi impressionnante. Même si la version locale utilise toujours la mémoire plus rapidement, il faut d'abord accéder plusieurs fois à la mémoire globale pour stocker les valeurs, nous nous attendions donc à ce que sa performance soit inférieure. Pourtant, il est possible que les cartes graphiques optimisent plusieurs copies de valeurs vers la mémoire locale, ce qui peut rendre les affectations plus efficaces que les recherches plus tard.

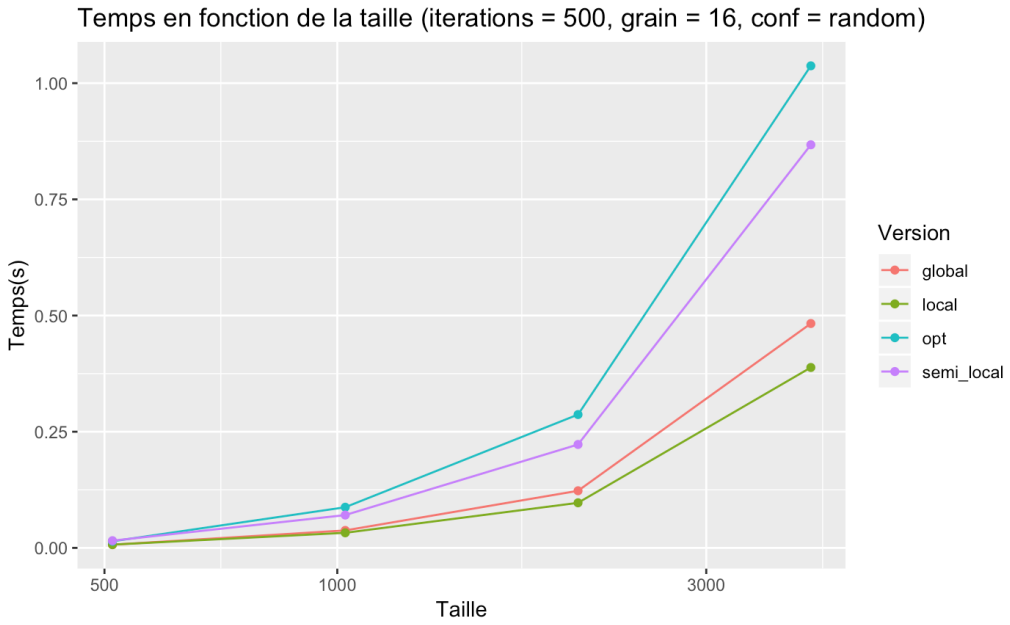

FIGURE 14 – Temps en fonction de la taille avec configuration *random* en *OpenCL*

Ensuite, avec la configuration *guns*, la version optimisée montre ses gains de performance, lorsque nous avons beaucoup de tuiles qui ne changent pas à chaque itération. Pour les autres implémentations, la relation entre elles continue la même.

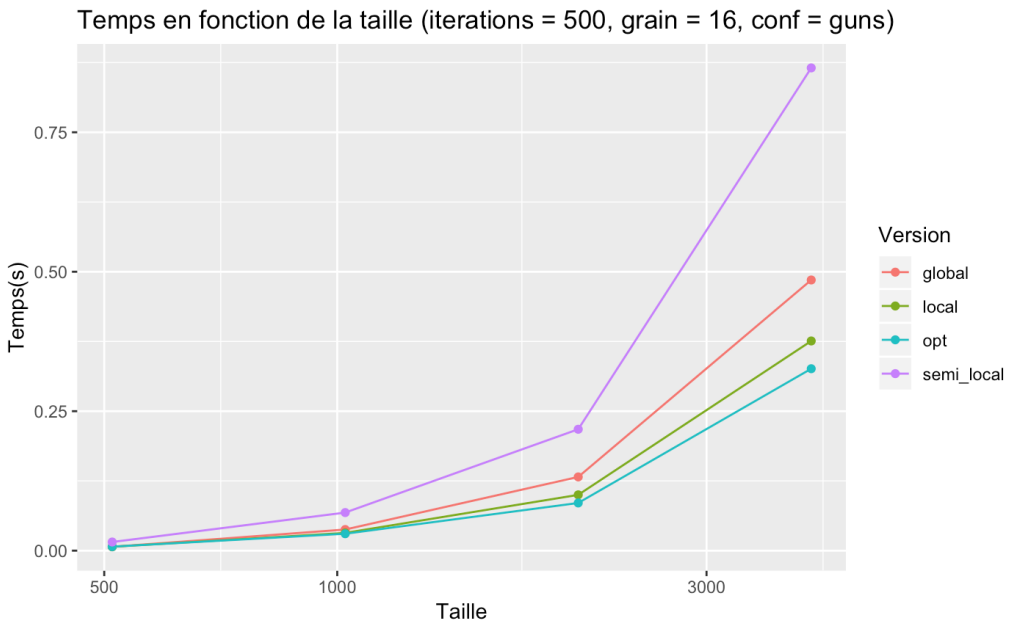

FIGURE 15 – Temps en fonction de la taille avec configuration *guns* en *OpenCL*

Comme pour la version basique, pour vérifier les effets du grain, nous regardons l'influence du grain. Nous avons choisi 2048 comme taille pour ces tests.

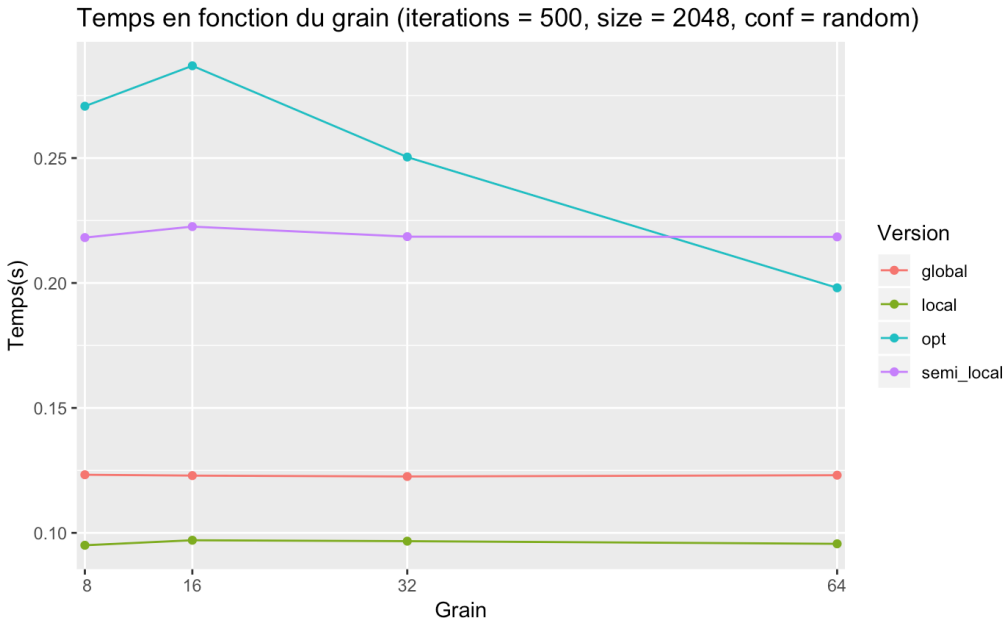

FIGURE 16 – Temps en fonction du grain avec configuration *random* en *OpenCL*

Pour la version optimisée, logiquement, des grains grands font qu'elle devient plus intéressante, mais pour les autres cela ne change pas beaucoup. Avec 32 grains nous avons déjà une réduction d'environ 50% même avec la configuration *random*. Pourtant, c'est toujours moins efficace que tout simplement calculer toutes les tuiles sans un contrôle de modifications.

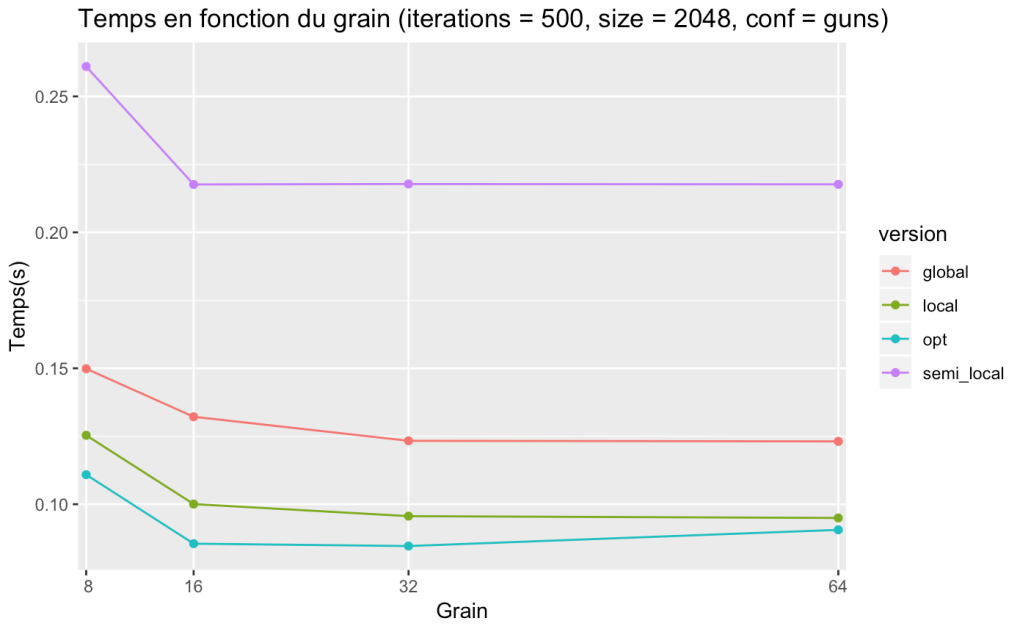

FIGURE 17 – Temps en fonction du grain avec configuration *guns* en *OpenCL*

Quand nous regardons le grain pour le motif initial *guns*, dès les plus petites valeurs nous avons déjà une exécution plus rapide avec la version optimisée mais le comportement n'est pas différent de celui de la version local. En fait, après 32 grains, l'optimisation commence à prendre trop de temps pour vérifier les matrices, d'autant que la version local continue à devenir plus rapide.

Pour conclure, nous montrons la différence entre l'implémentation la plus rapide et la plus lente.

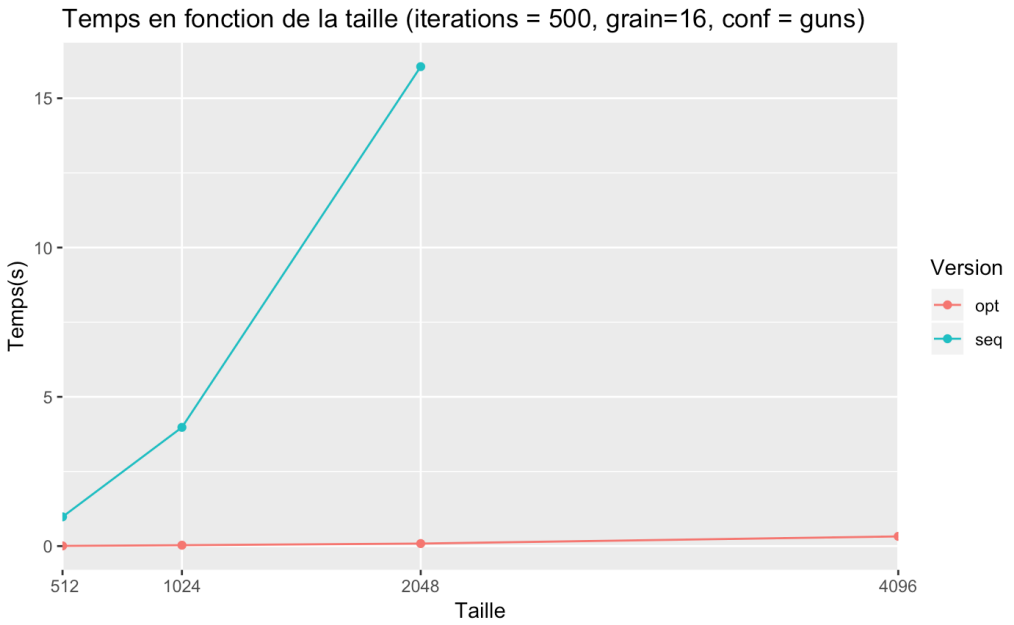

FIGURE 18 – Différence entre la version séquentielle et la version optimisée en *OpenCL*

7 Conclusion

Comme on peut le voir sur l'image 18, les gains que nous pouvons avoir en faisant une bonne implémentation peuvent changer complètement le temps d'exécution.

Pour ce projet l'effort pour réaliser un code parallèle a été considérable, mais il faut prendre en compte que c'était la première fois que nous travaillons sur une implémentation comme celle-ci.

Pour une prochaine fois, l'effort serait moins grand et donc le groupe croit que les gains de performance en général l'emportent largement sur l'effort.