Programmation Parallèle

Rapport final de projet

Clément Bertin
Emeric Duchemin
Laurent Genty

Sous la responsabilité pédagogique de
Raymond NAMYST

Année 2019-2020 - Semestre 8

1 Introduction

Dans ce projet nous souhaitons optimiser un algorithme d'automate cellulaire, le jeu de la vie ou game of life. Dans le sujet, nous avions une version dans laquelle, si une case est éteinte et qu'elle a 3 cases autour d'elle allumées, alors elle reste allumée. Si elle a 2 ou 3 voisines allumée exactement elle reste allumée. Nous avons essayé plusieurs options pour accélérer le code de complexité linéaire en la taille de l'image en partant d'une version séquentielle de base.

2 Optimisations implémentées

Dans ce projet, nous nous intéressons donc au noyau `life` . Nous allons partir de la version séquentielle de base fournie dans le Git afin de comparer les différentes versions implémentées et donc de connaître l'accélération totale par rapport à la version de base pour un programme classique :

— taille de fenêtre : 2048

— taille de tuile : 128

— nombre d'itérations : 1000

— pas de parallélisation

— aucune instructions AVX

— pas de calcul paresseux

2.1 Amélioration version tuilée

2.1.1 Gestion des bords

Avant toute chose nous avons effectué une amélioration préliminaire concernant la version tuilée de base. En effet, la version de base faisait une double boucle sur toutes les tuiles du jeu et ensuite lançait le traitement de la tuile : pour chaque pixel de la tuile, nous allons voir son état actuel et le changer en fonction des règles du jeu de la vie. Durant ce traitement, toutes les tuiles vont exécuter pour tous leurs pixels la fonction `compute_new_state` qui va potentiellement changer l'état du pixel. En revanche, dans cette dernière, nous regardons si le pixel se trouve en dehors de la fenêtre car nous effectuons des tests pour chacun de nos pixels.

Nous avons donc fait le choix d'effectuer le traitement tuilé des pixels comme avant mais **désormais**, nous excluons les bords. De fait, le traitement pour chaque pixel dans toutes les tuiles, n'a plus besoin d'effectuer le test sur les bords et cela nous sauve du temps CPU qui était utilisé afin de faire les tests pour les bords.

Ce traitement des bords est géré de la même manière (par une double boucle) et sera effectué juste après tout le traitement de la fenêtre.

2.2 Version OpenMP

La version `OpenMP`, nous permet de faire de la programmation parallèle multicoeur. Le but étant de paralléliser le traitement des tuiles, nous avons changé le code de la version `life_compute_omp_tiled` .

Comme nous l'avons dit précédemment, le code de base effectuait une double boucle sur les indices des tuiles et il s'agit là d'une partie de code très facilement parallélisable.

Nous avons utilisé une directive `pragma` autour de cette double boucle afin de passer à un pseudo-code comme suivant :

```
1 ||    #pragma omp parallel for schedule(dynamic)
2 ||    for (int x = 1; x < NB_TILE-1; x += 1) {
3 ||        #pragma omp parallel for schedule(dynamic)
4 ||        for (int y = 1; y < NB_TILE-1; y += 1) {
```

```
            if (!case_bord)
                do_tile(...);
        }
    }
```

Cette version nous permet de paralléliser le traitement de la boucle en effectuant en même temps les différentes lignes et colonnes de tuiles.

Cependant, cette version peut encore être améliorée, en effet, il s'agit là d'une double boucle avec `NB_TILE * NB_TILE` couples d'indices `x,y` . De fait, utiliser la directive `#pragma collapse(2)` nous permet de déplier l'espace d'itérations et donc de pouvoir créer `NB_TILE * NB_TILE` *threads* au maximum.

Nous passons donc à la version suivante :

```
    #pragma omp parallel for collapse(2) schedule(dynamic)
    for (int x = 1; x < NB_TILE-1; x += 1) {
        for (int y = 1; y < NB_TILE-1; y += 1) {
            if (!case_bord)
                do_tile(...);
        }
    }
```

Cette version nous améliore nos performances dans la mesure où notre espace d'itération sera unique désormais et donc plus facilement parallélisable.

À cela, nous ajoutons la parallélisation de l'amélioration précédente à savoir la gestion des bords. L'entièreté de la fenêtre est parallélisée avec `#pragma omp parallel for collapse(2)schedule(dynamic)` et ensuite nous gérons les bords de manière parallélisée aussi.

2.2.1 Choix de *scheduler*

Nous avions plusieurs choix concernant l'ordonnancement des *threads* sur les tuiles :

— `static` : attribution statique au départ selon un nombre de tuiles à prendre

— `dynamic` : dès lors qu'un *thread* a terminé sa tâche, il prend une tuile non traitée

— `guided`

Dans notre projet actuel, l'environnement est voué à évoluer au cours du temps : chaque *thread* aura potentiellement du travail à faire, mais dans une optique de calcul paresseux, il est possible que beaucoup de tuiles ne soient pas calculés. De fait, prendre un ordonnancement statique est peu pertinent : une fois l'attribution des tuiles aux *threads*, certains de ces derniers n'auront aucun calcul à faire et donc ... attendront tout simplement l'itération suivante. De plus, prendre un ordonnancement `guided` n'est pas optimal dans la mesure où nos opérations parallélisables ne sont pas précédées par d'autres opérations parallélisables avec une clause `#pragma omp sections nowait` (on ne profite donc pas du fait que les threads peuvent arriver à des moments différents à un for construire avec chaque itération nécessitant environ la même quantité de travail). La bonne chose à faire est donc de changer la politique d'ordonnancement et de garder celle dynamique permettant de faire en sorte d'attribuer les *threads* seulement dans les zones où il y a du travail et de ne pas faire attendre des *threads* pour rien au milieu de la fenêtre par exemple.

2.2.2 Comparaison des performances

Nous avons réalisé une courbe qui permet de se rendre compte de la performance suivant le nombre de coeur et de la rapidité des différentes configurations que nous avons précédemment présentées. Nous remarquons que la meilleure courbe est la courbe omp_tiled avec un schedule dynamique de 8, et qu'elle semble être optimale pour 24 coeurs. Lors de nos tests nous avons remarqué qu'avec notre implémentation, il y avait un léger bon aux alentours du 24ème coeur utilisé. Nous pensons donc qu'il

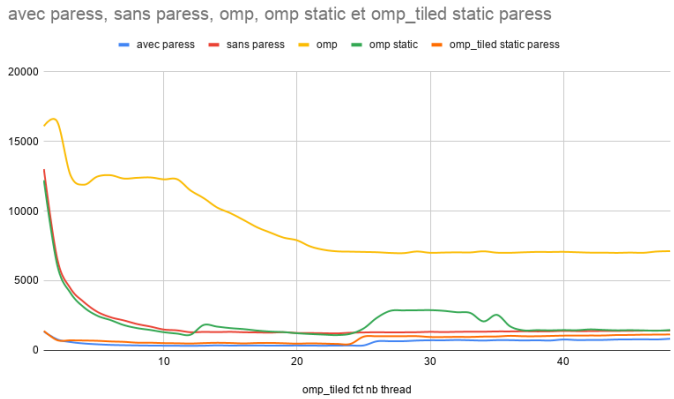
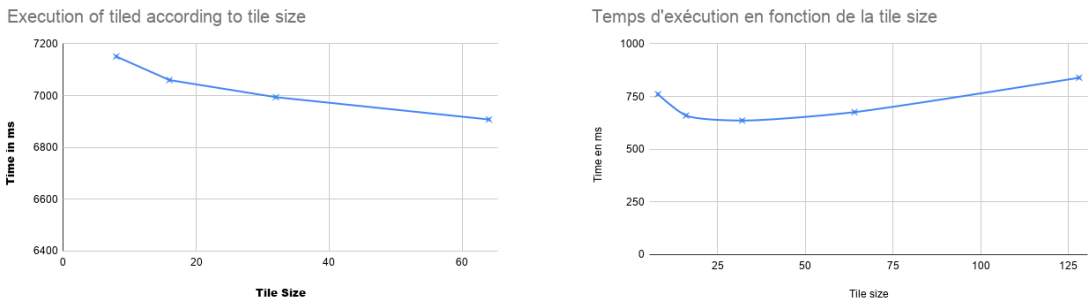


FIGURE 1 – Effets du nombre de coeurs utilisés sur les performances de différentes configurations

peut s'agir à la fois d'une difficulté d'ordonnancement, d'accès mémoire ou encore peut être le passage de coeurs physiques aux coeurs logiques. Nous observons tous ces résultats sur la courbe 1 :

Nous avons aussi essayé de tester quelles tile size était la meilleure pour notre algorithme de omp_tiled. C'est pourquoi nous avons mesuré nos performances en fonctions de la tile size. Nous obtenons donc que les tiles size 16 et 32 semblent être optimales comme le montre la figure 2.



(a) Temps pris pour la version séquentielle selon la taille des tuiles

(b) Effets de la tile size sur les performances de omp_tiled

FIGURE 2 – Temps pris pour la version parallélisée selon la taille des tuiles

On peut voir qu'on gagne un facteur 10 concernant le temps pris pour l'exécution et donc il s'agit d'une amélioration importante.

2.3 Évaluation paresseuse

2.4 Principe

Le but de l'évaluation paresseuse est de calculer uniquement les tuiles dans lesquelles il existe un pixel ayant une possibilité de changer d'état à la prochaine itération. Pour ce faire, nous utilisons un tableau de booléens représentant les tuiles ayant subit un changement à l'itération précédente. En effet, une tuile pour laquelle, elle-même et aucun de ses voisins n'a changé dans l'itération précédente ne peut pas subir de changement dans la prochaine itération. On enregistre donc à chaque itération quelle tuile à subit un changement. Ensuite, au moment de calculer la prochaine itération, on vérifie dans le tableau

FIGURE 3 – Comparaison des performances OMP avec et sans version paresseuse

de booléen si on a besoin de calculer la tuile. Pour ce faire on regarde si à l'itération précédente, la tuile ou une de ces voisines a subit un changement. Si oui on calcule la tuile. De cette manière, on réalise beaucoup moins de calculs mais on perd un peu de temps dans le calcul de l'évaluation paresseuse.

2.5 Comparaison des performances

Sur la figure 3, on peut voir qu'on compare les différentes versions implémentées :

— OMP-tiled sans paresseuse dynamic,8 (rouge)

— OMP-tiled avec paresseuse dynamic,8 (bleue)

— OMP simple dynamic,1 (jaune)

— OMP simple static,1 (vert)

— OMP-tiled avec paresseuse static,1 (orange)

Comme on peut le voir, la version la plus optimale est la version bleue : avec calcul paresseux et un schedule dynamic,8. Comme nous l'avons dit précédemment, attribuer dynamiquement les paquets de *threads* est plus optimal dans notre situation et nous permet d'atteindre 200ms par rapport à la version initiale séquentielle de 5500ms ce qui équivaut à une accélération de 27,5.

3 Vectorisation

3.1 Possibilités de vectorisation

Dans cette version, nous avons voulu optimiser les opérations facilement vectorisables : la somme autour de chaque pixel afin de connaître son nombre de voisins. Pour cela, nous devions changer nos structures de données : il n'était pas nécessairement utile d'utiliser des cellules codées sur 32 bits afin de manipuler des booléens. C'est donc pourquoi nous avons fait le choix de changer ces données en `char` (codés sur 8 bits).

Nous avons donc changé notre `cell_t` de `unsigned` (codé sur 32 bits) à un `char`. De fait, nos opérations sont désormais beaucoup plus facilement vectorisable. Nous avions donc désormais plusieurs choix de vectorisation :

— vectoriser la somme des cases voisines de chaque pixel

— vectoriser la somme des cases d'une tuile entière

Nous avons commencé par la première amélioration, nous avons donc décidé d'utiliser les méthodes AVX et notamment le type `__m256i`. Dans la fonction `compute_new_state_omp()` nous vectorisons la somme des 8 cases adjacentes. En effet, utiliser des fonctions AVX nous permet de faire un gain de performance non négligeable.

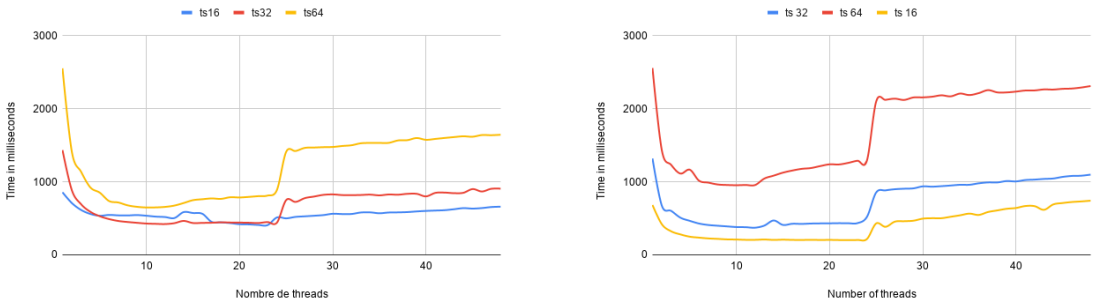
Le but étant de sommer la présence ou non de cellules dans les cases voisines et d'extraire cette valeur, nous arrivons donc au code suivant :

```
1 |     /* Vecteur nul */
2 |     __m256i m = _mm256_setzero_si256();
3 |     for (int i...) {
4 |         for (int j...)
5 |             /* Diffuser la valeur de la case courante i,j dans tous les vecteur m */
6 |             m = _mm256_add_epi8(m, _mm256_set1_epi8(cur_table (i,j)));
7 |     }
8 |     /* Recupere le nombre de voisins en extrayant la valeur au premier indice */
9 |     int n = _mm256_extract_epi8(m,0);
```

Comme nous pouvons le voir, au lieu de faire une somme classique dans un entier, nous allons donc stocker la somme des 8 cases voisines dans le vecteur `__m256i m` et ensuite nous allons extraire l'entier à l'indice 0 du vecteur `m`. On gagne ainsi du temps sur la somme des éléments. De fait, nous avons `n` le nombre de voisins vivants de manière beaucoup plus optimisée.

Concernant la deuxième méthode, à savoir vectoriser les tuiles entières, nous nous sommes heurtés à différents problèmes. En effet, la taille maximale d'un vecteur `__m256i` est de 32. En prenant en compte qu'une cellule est codée sur un `char`, cela nous ferait donc des vecteurs avec maximum 32 cellules. Cependant, cette taille est très petite dans la mesure où supposons que nous voulons travailler sur une taille de tuiles de 4, cela équivaut à stocker les 4*4 cellules à l'intérieur de la tuile, soit 16 cellules **mais aussi** les cases adjacentes qui ne se trouvent pas dans la cellule. En effet, la double boucle permettant d'avoir le nombre de voisins pour chaque cellule va de `x-1` à `x+1` (de même pour les `y`). Par conséquent cela reviendrait à avoir des données non pas de 4*4 mais de 6*6 ce qui est déjà trop grand pour nos vecteurs AVX `__m256i`.

3.2 Comparaison de performances



(a) Performances sans vectorisation AVX selon le nombre de threads sur 1000 itérations modèle de base

(b) Performances avec vectorisation AVX selon le nombre de threads sur 1000 itérations modèle de base

FIGURE 4 – Comparaison entre sans et avec vectorisation pour 1000 itérations avec le modèle de base

Comme nous pouvons le voir sur la figure 4, utiliser la vectorisation augmente nos performances. En effet, quand on choisit une bonne taille de tuiles (en l'ocurrence une taille de 16) on voit que l'on passe d'environ 500ms avec 24 threads sans vectorisation à 250ms avec 24 threads avec vectorisation. De

plus, on peut voir que lorsqu'on dépasse le nombre de coeurs physiques de la machine (24) on reduit nos performances considérablement. Par conséquent, augmenter constamment le nombre de *threads* n'augmente pas forcément les performances.

4 Open-CL

4.1 Implémentation

Nous avons implémenté une version du jeu de la vie en open-cl. Nous avons dû coder les fonctions d'initialisation (`life_init_ocl`), de dessin (`life_draw_ocl`) et d'appel de noyau (`life_invoke_ocl`) pour pouvoir avoir non pas 2 paramètres mais 4 paramètres pour une raison que nous verrons plus tard. Nous avons dans cette version implémenté de manière extrêmement simple le calcul des cases. Nous vérifions l'endroit de la case (est une case de coin, une case de bord, une case plus centrale), puis nous réalisons ensuite le calcul adéquat (afin d'éviter les segfaults =)). Nous avons dû aussi utiliser une matrice de correspondance afin d'avoir directement la matrice de sortie coloriée. On avait deux choix, soit directement mettre les couleurs dans les cases, soit appeler la fonction `life_refresh_img_ocl` . Nous avons pris le choix deux car il nous permettait d'éviter des calculs (et aussi parce qu'elle était super bien écrite ;)), mais il reste toujours la possibilité de le faire sans. Nous avons aussi essayé de minimiser le temps passé par chaque pixel dans les enchaînement de if. C'est pourquoi nous avons imbriqué le plus possible de else afin d'éviter des vérifications inutiles. Cependant des pistes d'amélioration de cette version seraient d'éviter les conditionnelles if, par exemple en allouant un tableau plus grand, comme ce que nous avons proposé en commentaire pour la version paresseuse.

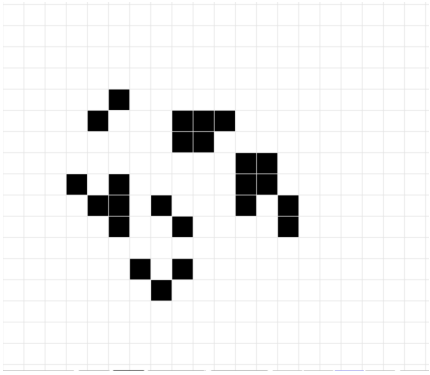
4.2 Comparaison de performances

Dans cette version nous avons eu une surprise, **notre** version paresseuse était moins rapide que la version classique (nous avons préféré mettre en commentaire cette version du fait de ses performances plus faibles). Nous l'expliquons de la façon suivante : la version avec paresseux rajoute des phases de tests en plus du calcul des cases adjacentes ainsi qu'un calcul qui réalise un `OU` binaire pour savoir si une des cases adjacentes doit être calculée. La raison pour laquelle nous avons 4 paramètres est que l'on doit stocker le tableau recensant les cases devant être calculées dans la version paresseuse. Ce calcul et ce test sont au final très légèrement moins rapide pour un groupe par rapport au simple calcul de l'état à l'étape suivante. Nous avons essayé d'optimiser au maximum la version paresseuse en évitant toute forme de `if` (il aurait fallu donc allouer un tableau plus grand que nécessaire pour permettre des vérifications dans des tuiles non "réelles"). Nous pensons cependant qu'une version paresseuse plus optimisée pourrait avoir des performances encore meilleures. Nous obtenons même sans évaluation paresseuse des résultats que nous jugeons plutôt bon à savoir que nous obtenons une accélération supérieur à 170 sur notre code pour l'exécution de `./run -k life -i 1000 -s 6208 -a meta3x3 -n -v seq` . Nous avons aussi vérifié que qu'il s'agit du bon résultat qui est calculé : par exemple pour *diehard* avec 50 itérations nous obtenons la figure 5 :

Et pour *diehard* 100 nous avons la figure 6 :

5 Conclusion

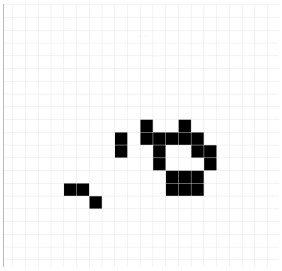
Dans ce projet nous avons réussi a utiliser la plupart des notions apprises dans la matière : la programmation parallèle, la vectorisation, la programmation sur carte graphique... Cependant, toutes les améliorations ne se valent pas. L'accélération maximale pertinente que nous ayons obtenu est de 170 en passant de la version séquentielle fournie de base à celle optimisée en `OpenCL` concernant le modèle `meta3x3` de taille 6208.



(a) Avec l'algorithme de Wikipédia pour 50 itération

(b) Avec le programme en version opencl pour 50 itération

FIGURE 5 – Comparaison entre les résultats de opencl et de wikipédia pour 50 itérations



(a) Avec l'algorithme de Wikipédia pour 100 itération

(b) Avec le programme en version opencl pour 100 itération

FIGURE 6 – Comparaison entre les résultats de opencl et de wikipédia pour 100 itérations

Cette accélération nous démontre parfaitement que derrière un programme simple, il existe des méthodes afin d'améliorer considérablement les performances. Que l'on passe par de la programmation multicoeur ou bien de la programmation sur carte graphique, nous pouvons toujours améliorer les performances d'un programme (jusqu'à un certain point tout de même).

Dans notre cas, nous aurions pu encore améliorer les performances de notre noyau dans la version `OpenCL` si nous avions implémentée la version paresseuse. En effet la version actuelle calcule toutes les tuiles, cependant, nous l'avions vu durant notre projet et dans ce rapport, que calculer seulement les tuiles qui ont été changées au temps t - 1 nous permettait de faire un grand gain en performances. La dernière Figure 7 résume finalement plutôt bien les performances observées :

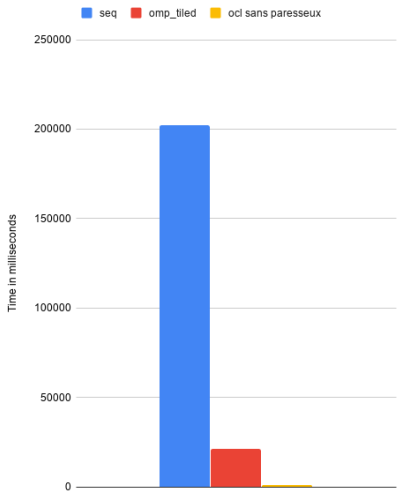


FIGURE 7 – Benchmark du meilleur des versions omp tuilées, séquentielle et ocl sur le jeu meta3x3 de taille 6208