Filière Informatique

Projet GPU

Optimisation multi-threadée & vectorielle

Tanguy Pemeja
Maxime Rosay

Encadrant :
Raymond Namyst

Version du 20 Mai 2020

Introduction

L'idée de ce projet est d'utiliser différentes techniques pour optimiser le calcul des itérations du jeu de la vie. Selon la configuration, ce jeu peut devenir très coûteux en termes de ressources, notamment mémoires.
L'objectif est le suivant: à partir de la version séquentielle de ce dernier, il faut produire différentes versions du code mettant en avant diverses optimisations afin d'augmenter la rapidité d'exécution de chaque itération.

1 Version de base

Nous n'allons pas revenir sur la version séquentielle déjà implémentée dans `EASYPAP` visible dans `life.c` mais nous allons directement aborder les optimisations réalisées.

1.1 Utilisation de Open MP

Pour paralléliser n'importe quel type de code rapidement, une optimisation naturelle[1] est d'utiliser les macros `pragma omp` devant les boucles `for`.
C'est ce que nous avons fait dans un premier temps, en rajoutant aussi le mot clé `collapse(2)` permettant d'optimiser les tâches à distribuer sur deux boucles `for` consécutives.
Cependant, il subsiste une section critique qui est le retour de la fonction `do_tile` contenu dans la variable `change` permettant d'indiquer si à cette itération un changement a eu lieu sur l'ensemble des cellules. Pour palier à cela, nous avons utiliser une variable privée à chaque thread ainsi qu'une réduction sur l'instruction `|` permettant d'éviter la concurrence sur le calcul de la variable change à la fin de chaque itération.
A noter que l'instruction `single` certifie que la ligne suivante sera exécutée par un seul thread.
Nous obtenons alors le code suivant:

```
1  unsigned life_compute_ompfor(unsigned nb_iter) {
2    unsigned res = 0;
3    unsigned change = 0;
4    unsigned changetmp = 0;
5  #pragma omp parallel firstprivate(changetmp)
6    for (unsigned it = 1; it <= nb_iter; it++) {
7
8  #pragma omp for collapse(2) reduction(| : change) schedule(runtime)
9      for (int y = 0; y < DIM; y += TILE_SIZE) {
10       for (int x = 0; x < DIM; x += TILE_SIZE) {
11         changetmp |=
12             do_tile_ompfor(x, y, TILE_SIZE, TILE_SIZE, omp_get_thread_num());
13         change = change | changetmp;
14       }
15     }
16 #pragma omp single
17     swap_tables();
18     if (!change) { // we stop when all cells are stable
19       res = it;
20       break;
21     }
22   }
23   return res;
24 }
```

[1] A condition d'avoir suivi le cours IT224

1.2 Résultat

Les courbes observables en figure 1 ont été générées au CREMI avec la commande suivante:

```
OMP_SCHEDULE=static ./run -k life -s 2048 -i 1000 -ts 32
```

En faisant varier le nombre de coeurs utilisés, et en fixant une taille de tuile à 32[2].

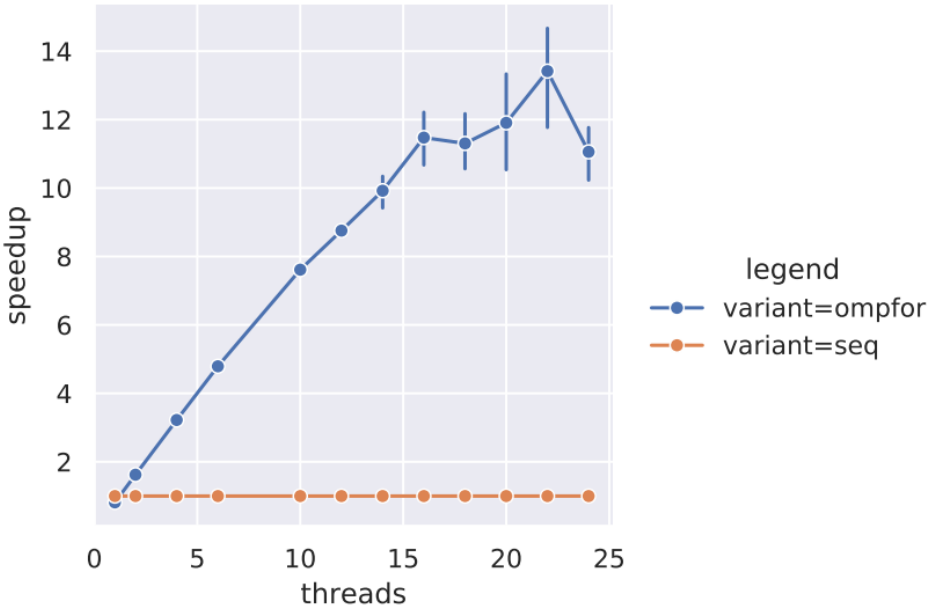

Figure 1: Comparaison des versions séquentielles et parallélisées

Nous observons ainsi clairement le gain de la version parallélisée par rapport à celle séquentielle. En utilisant x threads au lieu d'un seul nous devrions avoir en théorie un gain maximal de x, cependant on remarque que l'on est en dessous pour chaque point sur les courbes. Ce qui est normal puisque plus il y a de threads en parallèle, plus les dépendances de la rapidité des coeurs du CPU augmentent (si un coeur se ralentit, cela pénalise tout les autres).

Ainsi, se contenter seulement de la figure ci-dessus n'est pas suffisant, poussons le test plus loin en allant jusqu'à utiliser les 48 coeurs disponibles au CREMI ainsi qu'en comparant les versions statiques et dynamiques.
Les résultats de la figure 2 ont été générés avec la même commande que précédemment, cependant en plus du nombre de threads variable, nous avons aussi l'ordonnancement des tâches Open MP:

[2]32 est optimale, les autres étant moins optimisées

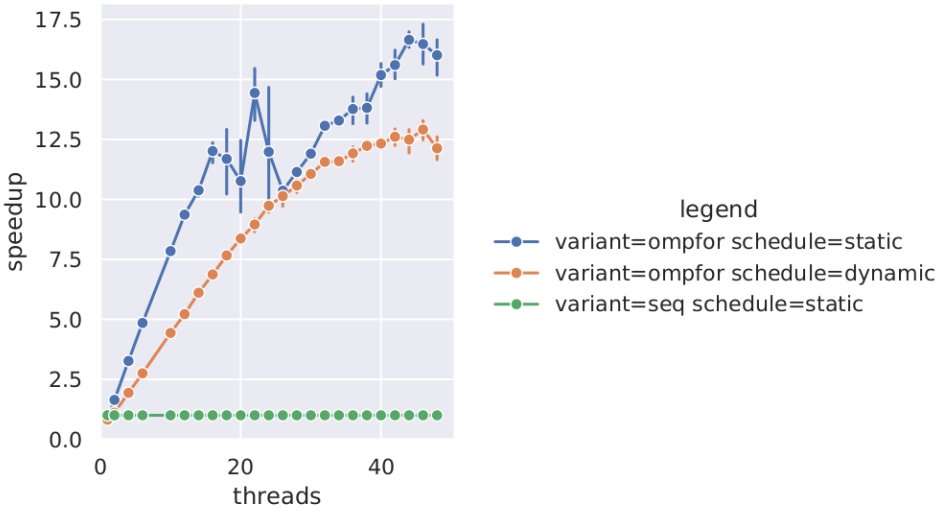

Figure 2: Comparaison des versions dynamiques et statiques

Comme attendu, la version dynamique est nettement en dessous de celle statique, puisque cette fois ci, les coeurs sont tous actifs dès qu'il y a du travail, cependant, la mémoire est limitante ici. En effet, nous avons beaucoup plus de `cache miss` puisque les threads vont prendre des morceaux de la figure à calculer à des endroits divers et variés et non dans une zone prédéfinie. Cette version dynamique est certes beaucoup moins dépendantes de la disponibilité[3] des coeurs mais la mémoire représente un réel frein.

2 Version avancée

Dans cette section nous allons aborder des optimisations plus recherchées que seulement utiliser Open MP afin de paralléliser le code. L'idée est de limiter les accès mémoires coûteux et de passer par des vecteurs pour représenter les cellules. A noter que nous ajoutons les optimisations décrites dans la suite à la version Open MP.
La seule modification notable est que dans la fonction itérative `life_compute` nous ne calculons pas les cellules en bordure, ie. pour $x = 0$,$y = 0$,$x = DIM - 1$ et $y = DIM - 1$

2.1 Optimisation vectorielle

L'objectif de cette partie est d'optimiser le calcul de vie ou de mort. L'idée est de réaliser ce calcul en parallèle grâce à des vecteurs de 256 bits. Un tel vecteur nous permet de représenter 32 chars et donc 32 cases. Mais la difficulté majeure d'une telle démarche est d'utiliser des calculs vectoriels afin d'atteindre le même objectif. Ici, la première étape de notre démarche est de calculer le nombre de voisin de ces 32 cases.

2.1.1 Chargement des données

La première étape consiste à récupérer les valeurs de vie ou de mort de notre vecteur ainsi que de leurs voisins.

```
1  static int do_tile_reg_256(int x, int y) {
2    int change = 0;
3
4    __m256i haut, milieu, bas, somme, next_i, n, gauche, droit;
5
```

[3] On entend utilisation ici

```
6     haut = _mm256_load_si256((const __m256i *)&cur_table(y - 1, x));
7     milieu = _mm256_load_si256((const __m256i *)&cur_table(y, x));
8     bas = _mm256_load_si256((const __m256i *)&cur_table(y + 1, x));
9     #Chargement des données
10
11    if(x > 0) {
12      completion_gauche =
13          cur_table(y - 1, x - 1) + cur_table(y, x - 1) + cur_table(y + 1, x - 1);
14    }
15    if (x + 32 < DIM) {
16      completion_droite = cur_table(y - 1, x + 32) + cur_table(y, x + 32) +
17                          cur_table(y + 1, x + 32);
18    }
19    #Récupérer les informations manquantes
```

Comme on peut le voir ci-dessus dans le code, nous chargeons les données des voisins pour le calcul des cases de la ligne y et les colonnes $[\![x; x + 32]\!]$. Ces données sont composées du vecteur en lui-même (définit par la variable milieu dans notre code), les vecteurs représentants les cases au-dessus et en-dessous de ce dernier ainsi que des valeurs complémentaires afin de calculer ses extrémités si jamais nous n'atteignons pas les bords de notre fenêtre.

```
1     somme = _mm256_add_epi8(haut, _mm256_add_epi8(milieu, bas));
```

Cette ligne permet de calculer le nombre de voisin actif par colonne pour notre vecteur comme sur la figure 3.

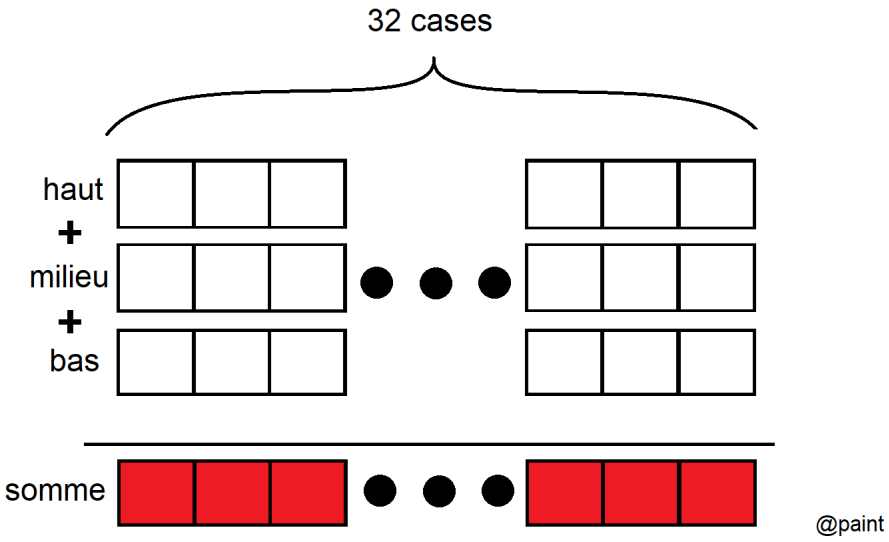

Figure 3: Calcul du nombre de voisin actif par colonne

Maintenant que nous avons ce vecteur qui représente le nombre de voisins par colonne, il faut additionner les bonnes cases afin d'obtenir pour chacun d'elle son nombre de voisins dans notre vecteur[4]. Pour se faire, nous effectuons un shift gauche et un shift droit de notre somme avant de sommer ces 3 vecteurs et ainsi obtenir le nombre total de voisin (conf figure 4 ci-dessous).

[4]ie. chaque case doit être la somme d'elle-même avec ses cases adjacentes (gauche & droite)

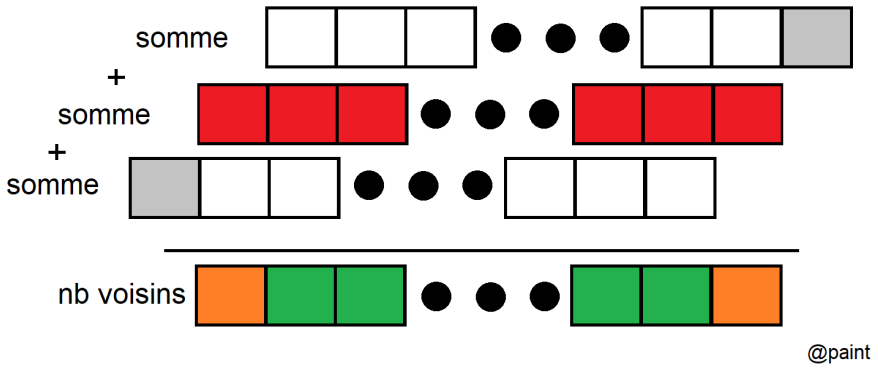

Figure 4: Calcul du nombre total de voisin

Sur la figure 4, les cases vertes correspondent à une valeur du nombre de voisins correct tandis que les 2 cases oranges correspondent à des valeurs partielles dû à une perte d'information lors des shifts.

Pour compenser cette perte, nous ajoutons les valeurs complémentaires recueillies lors du chargement des données.

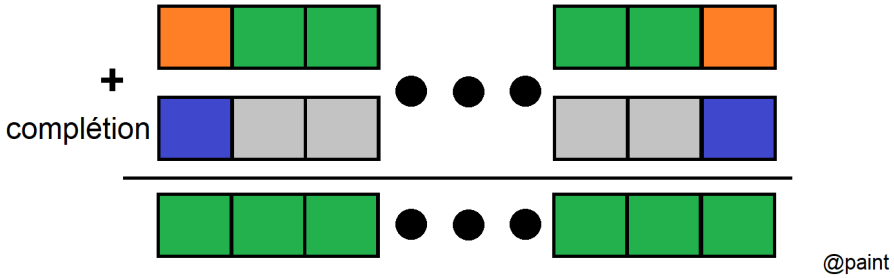

Figure 5: Ajout d'un vecteur de complétion afin de calculer les extrémités

```
int milieu_droit = _mm256_extract_epi8(somme, 16);
int milieu_gauche = _mm256_extract_epi8(somme, 15);
#Permet de compenser le shift par morceaux de 128 bits pour en faire un de 256 bits (32 cases)

int completion_gauche = 0;
int completion_droite = 0;


__m256i completion =
    _mm256_set_epi8(completion_droite, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, milieu_gauche, milieu_droit, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, completion_gauche);
#Complète afin d'obtenir toutes les informations pour les 32 cases (256 bits)

gauche = _mm256_slli_si256(somme, 1);
droit = _mm256_srli_si256(somme, 1);

n = _mm256_add_epi8(
```

```
19          somme, _mm256_add_epi8(completion, _mm256_add_epi8(gauche, droit)));
20      #Calcul du nombre totale de voisins
21  }
```

Le code ci-dessus reprend les principes décrits Figure 4 et 5 avec toutefois une légère différence. En effet, le shift (gauche par exemple) implémenté par Intel `_mm256_slli_si256` effectue un shift par bloc de 128 bits. C'est-à-dire que pour notre vecteur de 256 bits, cette fonction intrinsèque effectuera un shift multiple de 8 bits sur les 128 bits de poids faible puis sur les 128 bits de poids forts. Ainsi, en plus de perdre une information sur les extrémités, nous perdons deux valeurs supplémentaires. Si nous divisons notre vecteur de 256 bits en blocs de 8 bits (char) nous perdons les valeurs aux indices 15 et 16.
C'est pourquoi nous ajoutons ces valeurs dans le vecteur de complétion afin de préserver l'intégrité de notre calcul.
Maintenant que nous avons le nombre de voisins pour chacune de nos 32 cellules, nous devons alors effectuer un test de vie ou de mort sur ces dernières. Dans l'implémentation cela donne le code suivant:

```
1     n = (n == 3 + me) | (n == 3);
2     if (n != me)
3       change |= 1;
```

Que nous avons simplement adapté vectoriellement en :

```
1   next_i = _mm256_or_si256(
2         _mm256_cmpeq_epi8(n, _mm256_set1_epi8(3)),
3         _mm256_cmpeq_epi8(n, _mm256_add_epi8(_mm256_set1_epi8(3), milieu)));
4   next_i =
5         _mm256_and_si256(next_i, _mm256_set1_epi8(1)); // set à 1 au lieu de 255
6     change |= !_mm256_testz_si256(_mm256_set1_epi8(255), _mm256_xor_si256(next_i, milieu));
```

2.2 Ajout de la méthode de calcul paresseuse

Pour chaque vecteur de 256 bits (32 char) nous stockons dans un tableau annexe un booléen indiquant si à l'itération i il y a eu un changement. Cela permet à l'itération $i+1$ de tester si un de ses voisins a eu des changements des à l'étape i.

```
1   static int test_paresseux(int x, int y) {
2     if (x > 0 && x < DIM - 1) {
3       /* 9 itérations max*/
4       for (int i = y - 1; i <= y + 1; i++)
5         for (int j = x - 1; j <= x + 1; j++) {
6           if (change_table[index_change(i, j)] > 0) {
7             return 1;
8           }
9         }
10    }
11    return 0;
12  }
```

Il aurait été possible d'optimiser ce test en distinguant les différents changements. En effet, notre vecteur est influencé que par les changements de toutes les cases des vecteurs d'au-dessus et d'en-dessous. Et seules les extrémités des vecteurs de chaque côté pourra impacter notre vecteur. Cela permettrait d'augmenter le nombre de fois où l'on effectue l'attente paresseuse.

2.3 Résultat

Les courbes observables à la figure 6 ont été générées au CREMI avec la commande suivante sur 10 lancements par nombre threads:

```
1 OMP_SCHEDULE=static ./run -k life -s 2048 -i 1000
```

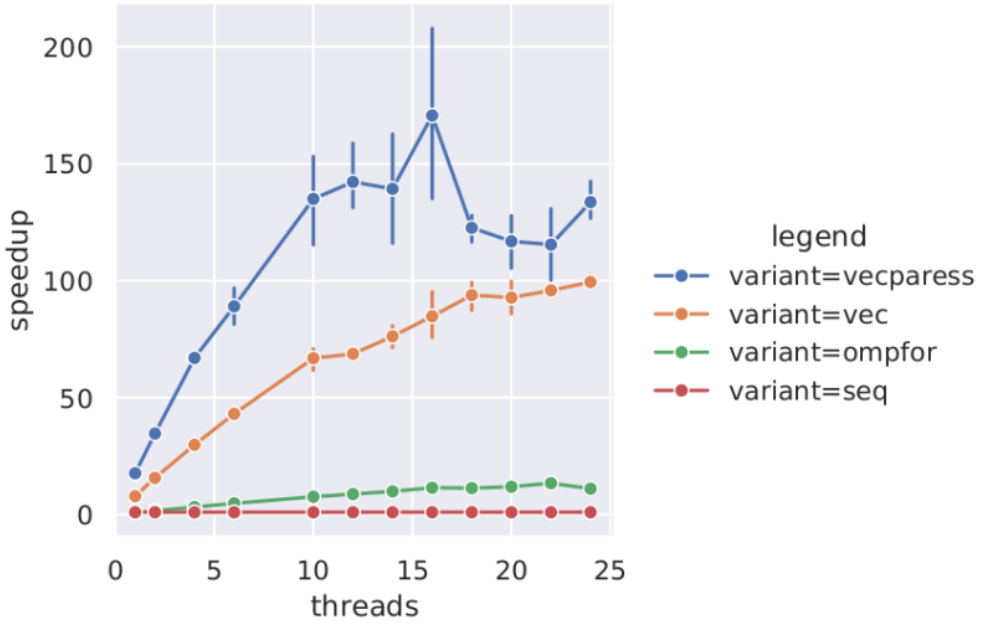

Figure 6: Comparaison de toutes les versions

On observe que plus on ajoute d'optimisations, plus l'exécution est rapide. Cependant, on remarque des pics ainsi qu'une baisse de performance pour la version vectorielle paresseuse dû au fait qu'il n'y ait pas assez de tâches à distribuer.
Comme pour la version Open MP classique, nous avons poussé les tests encore plus loin pour cette version. Les résultats sont observables dans la figure 7, en jouant avec la taille de la fenêtre à calculer pour la même commande que précédemment:

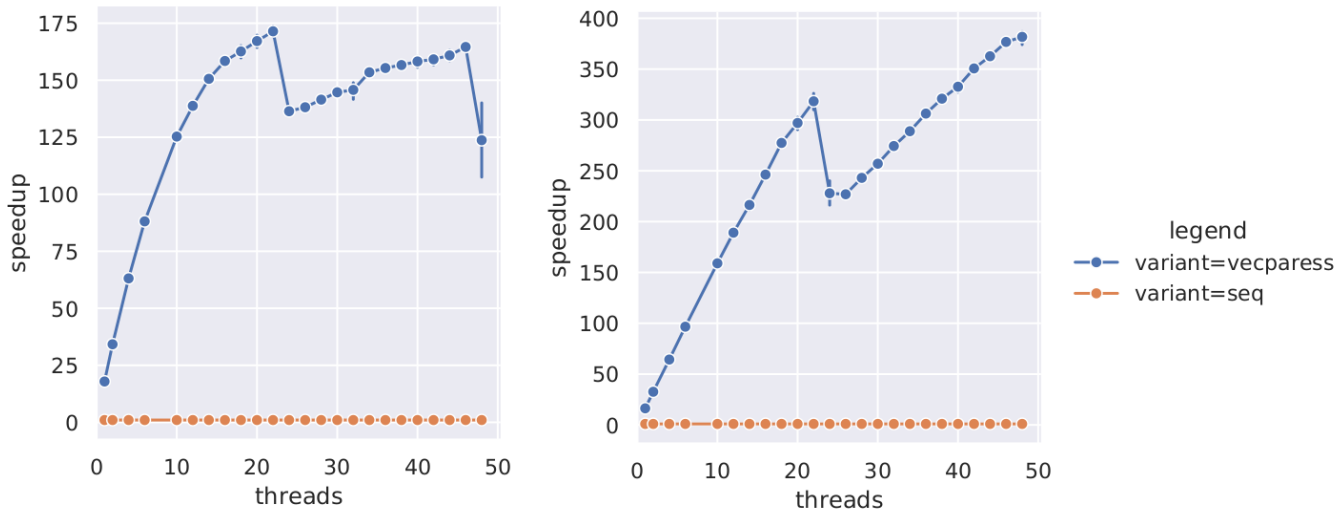

Figure 7: Comparaison des versions vectorielles paresseuses pour une taille 1024 (gauche) et 4096 (droite)

C'est dans la configuration la plus grande (4096) que la version vectorielle paresseuse prend sens, il y a beaucoup de travail et donc plus nous utilisons de coeurs plus les performances sont bonnes. Dans le cas contraire (1024), il n'y a pas une aussi grande charge de travail ainsi les performances sont moindres en fonction du nombre de threads[5].
On observe sur ces deux affichages qu'il y a un pic descendant lorsque nous avons 24 threads. Ceci est dû à la machine du CREMI qui possède 24 coeurs physiques et donc à partir de ce moment là, nous avons une baisse des performances puisque on utilise alors des coeurs virtuels. Ainsi, des coeurs physiques se retrouvent avec deux fois plus de travail à effectuer ![6]

[5]C'est comme avoir une RTX 2080 pour faire du traitement de texte
[6]Merci M. Namyst !