Rapport de Projet IT224

Parallélisation du Jeu de la Vie

Badr El Habbassi, Othmane Boutahri

Projet IT224
Département Informatique
S8 - Année 2024/2025

Table des matières

1 Introduction

Ce rapport présente notre travail dans le cadre du module IT224. L'objectif principal est de paralléliser le kernel `life.c` en utilisant différentes techniques d'optimisation sur CPU (notamment OpenMP, tiling et lazy evaluation), et d'implémenter une version GPU avec CUDA. Nous allons comparer les performances obtenues avec ces optimisations sur plusieurs jeux de données et explorer la portabilité des optimisations en fonction des tailles et configurations des données.

Le jeu de la vie, un modèle de simulation de cellules vivantes et mortes sur un damier, est utilisé comme base pour ce projet. Ce modèle est un exemple typique de calcul *stencil*, dans lequel chaque cellule dépend de ses voisines dans la grille.

Tous les tests ont été réalisés exclusivement sur les machines du CREMI. Les informations détaillées concernant la configuration matérielle utilisée sont disponibles en Annexe A 9.

2 Analyse du Code de Base

2.1 Fonctionnement du Jeu de la Vie

Le jeu de la vie est un automate cellulaire qui évolue à chaque itération en appliquant un ensemble de règles simples à un ensemble de cellules vivantes et mortes. Chaque cellule est entourée de huit voisines et, à chaque itération, son état (vivant ou mort) dépend de l'état de ses voisines selon les règles suivantes :

— Une cellule morte devient vivante si elle a exactement trois voisines vivantes, sinon elle reste morte.

— Une cellule vivante reste vivante si elle est entourée de deux ou trois voisines vivantes, sinon elle meurt.

Les règles sont connues sous le nom de `B3/S23` : "naissance si trois voisins" et "survie si deux ou trois voisins".

2.2 Structure du fichier `life.c`

Le fichier `life.c` contient plusieurs fonctions qui sont essentielles pour l'initialisation, la mise à jour des cellules, et la gestion des itérations. Voici un aperçu des principales fonctions utilisées :

— `life_init` : Initialise les structures de données et alloue la mémoire pour la grille du jeu.

— `life_do_tile_default` : Fonction utilisée pour appliquer les règles du jeu à une tuile de cellules.

— `life_compute_seq` : Implémente le calcul séquentiel des cellules, où chaque cellule est mise à jour de manière indépendante.

— `life_compute_tiled` : Implémente le calcul en utilisant le tiling, où les cellules sont traitées par blocs (tuiles), permettant une meilleure localité mémoire et un parallélisme plus efficace.

2.3 Pipeline d'exécution

Le pipeline d'exécution du programme suit les étapes suivantes :

1. `life_init` : Alloue et initialise les structures de données nécessaires pour simuler le jeu.
2. `life_do_tile_default` : Applique les règles du jeu à une portion de la grille (une tuile) dans l'implémentation séquentielle ou parallélisée.
3. `life_compute_seq` ou `life_compute_tiled` : Exécute les itérations en fonction de l'optimisation choisie (séquentielle ou parallélisée).
4. `life_finalize` : Libère la mémoire allouée et termine le programme.

Les fonctions sont appelées de manière séquentielle et parallélisée pour exploiter au mieux les ressources du CPU.

3 Implémentation OpenMP

3.1 Première version parallélisée

Nous avons commencé par paralléliser la fonction `life_compute_tiled` à l'aide de la directive `#pragma omp parallel for collapse(2)`. La variable `change` a été protégée par une clause `reduction(|:change)` afin d'éviter toute condition de concurrence. Cette première version constitue la base de référence OpenMP pour la suite de l'étude.

La version optimisée `life_compute_omp` parallélise les calculs du jeu de la vie en utilisant OpenMP. Grâce à la directive `#pragma omp parallel for collapse(2) reduction(|:change) schedule(runtime)`, les deux boucles imbriquées parcourant les tuiles de la grille sont parallélisées, ce qui améliore l'utilisation des cœurs processeur et la répartition de la charge de travail. La clause `reduction(|:change)` évite toute condition de concurrence sur la variable `change`, assurant la validité des résultats. Enfin, le choix dynamique de la taille des tuiles et de la politique de répartition permet d'optimiser les performances selon l'architecture de la machine.

Influence de la taille de tuile, du scheduling et du nombre de threads

Nous avons étudié l'influence des principaux paramètres d'OpenMP et du tiling sur les performances de la version parallélisée de `life_compute_tiled`. Les tests ont été effectués avec :

— une grille de taille `1024`,
— un jeu de données constant : `random`,
— une charge de travail constante, avec un nombre d'itérations calculé selon la formule :

$$\text{itérations} = \frac{100 \times 8192^2}{1024^2} = 6400$$

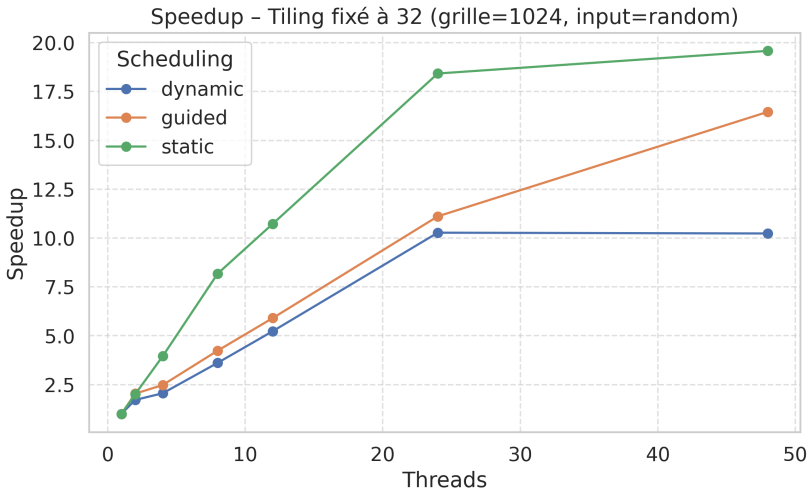


FIGURE 1 – Speedup obtenu selon le scheduling, pour une tuile de taille 32.

Variation du scheduling (tuile fixée à 32). Analyse : La stratégie `static` montre un gain presque linéaire jusqu'à 24 threads, puis continue de progresser légèrement jusqu'à 48, surpassant `dynamic` et `guided`. `Dynamic` plafonne autour de 10×, suggérant une surcharge de gestion ou une mauvaise répartition, tandis que `guided` se situe entre les deux. Le scheduling statique se distingue par sa répartition prévisible et uniforme des tâches, minimisant les coûts d'ordonnancement et les synchronisations. Contrairement aux stratégies `dynamic` et `guided`, il n'implique pas de gestion

dynamique des files de tâches, ce qui réduit les surcharges et améliore l'exploitation des caches L1 et L2. En assignant des blocs contigus à chaque thread, `static` optimise la localité mémoire et limite la contention, maximisant ainsi le parallélisme pour des charges de travail régulières. Pour des charges irrégulières, cependant, `dynamic` et `guided` peuvent offrir une meilleure répartition des tâches.

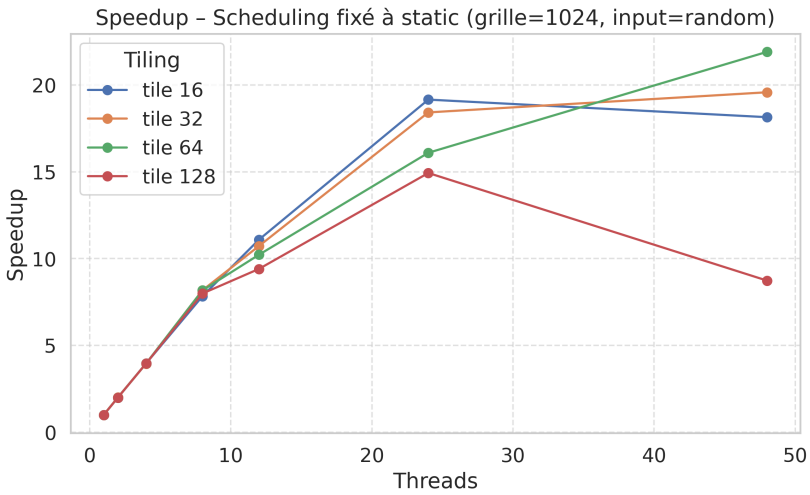


FIGURE 2 – Speedup obtenu en fonction de la taille de tuile avec scheduling `static`.

Variation de la taille de tuile (scheduling fixé à `static`). Analyse : Les tuiles de taille `64` donnent les meilleurs résultats au-delà de 24 threads. `32` est légèrement inférieure mais reste très stable. En revanche, `128` montre une forte régression à 48 threads, probablement due à une granularité trop faible (manque de parallélisme) ou à une mauvaise localité mémoire.

Interaction entre l'optimisation et les différents jeux de données

Pour cette analyse, nous avons étudié l'impact de l'optimisation (OpenMP et Tiling) sur les performances en faisant varier le jeu de données. Les tests ont été effectués avec :

- une taille de grille de `1024`,
- une taille de tuile de `32`,
- une stratégie de répartition (scheduling) `static`,
- les jeux de données `random`, `ship`, `moultdiehard130`, et `moultdiehard2474`.

Les résultats montrent comment l'optimisation affecte chaque jeu de données. Nous avons mesuré le speedup par rapport à l'exécution séquentielle et comparé les performances de chaque jeu de données pour une variation du nombre de threads.

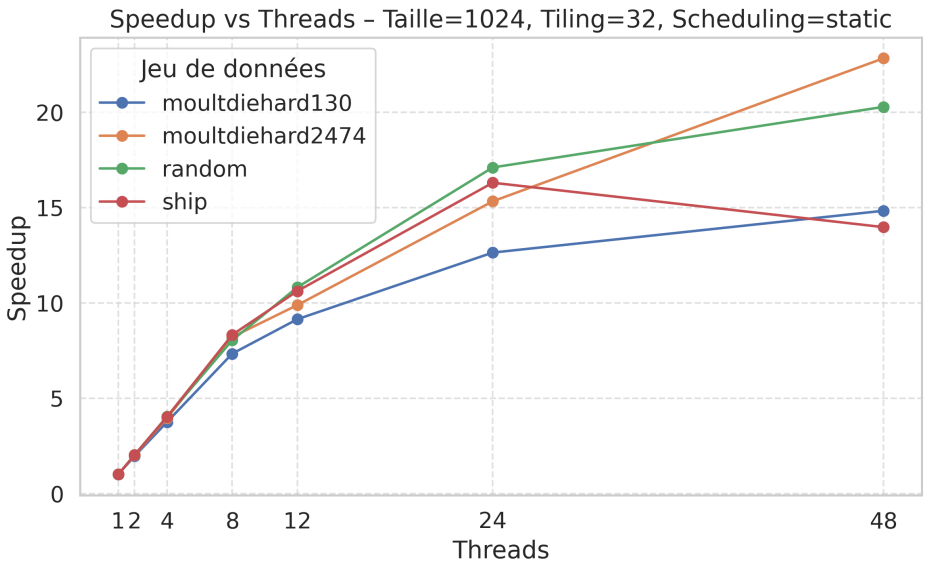


FIGURE 3 – Impact de l'optimisation OpenMP et Tiling pour différents jeux de données (`grille=1024, tile=32, schedule=static`)

Analyse :
- Le jeu de données `moultdiehard130` semble être le plus sensible à l'optimisation, avec des gains de speedup importants à partir de 12 threads.
- Le jeu `ship` bénéficie également de l'optimisation mais à un taux moindre.
- Le jeu `random` montre un comportement stable, avec un speedup qui augmente de manière linéaire jusqu'à 24 threads avant de plafonner.
- Le jeu `moultdiehard2474` montre un gain modéré, suggérant qu'il soit moins dépendant de l'optimisation que d'autres jeux de données.

4 Optimisations Avancées

4.1 Version Lazy

L'optimisation *Lazy Evaluation* repose sur une observation simple : dans de nombreuses configurations, certaines régions du domaine de calcul restent stables sur plusieurs itérations (aucune cellule ne change d'état). Il est donc inefficace de recalculer ces régions à chaque itération. Cette optimisation vise ainsi à ignorer le traitement des tuiles dont on peut garantir qu'elles ne subiront aucun changement lors de l'itération suivante.

Principe L'idée est de suivre les zones actives du jeu à l'aide de deux tableaux (`lazy_table_cur` et `lazy_table_next`) où chaque élément indique si une tuile a changé. À chaque itération, seules les tuiles ayant eu des changements dans leur voisinage lors de l'itération précédente sont recalculées. Cela permet de réduire considérablement le nombre de calculs réalisés, en particulier sur des configurations partiellement ou temporairement stables.

Implémentation Deux tableaux sont utilisés pour suivre les modifications de chaque tuile entre les itérations :
— `lazy_table_cur` : Table représentant les changements de l'itération courante.
— `lazy_table_next` : Table pour l'itération suivante.

Chaque élément de ces tableaux indique si la tuile correspondante a été modifiée. Une tuile est recalculée uniquement si elle ou l'un de ses voisins a changé lors de l'itération précédente. Cette logique est implémentée dans la fonction `tile_has_active_neighbor()`.

Code de base L'implémentation combine cette stratégie avec OpenMP pour conserver le parallélisme. Le code ci-dessous montre l'initialisation, le cœur de la boucle d'itération, ainsi que les fonctions de gestion des tableaux :

```
static unsigned *restrict lazy_table_cur = NULL;
static unsigned *restrict lazy_table_next = NULL;

#define lazy_cur(y, x) lazy_table_cur[(y) * TILE_X + (x)]
#define lazy_next(y, x) lazy_table_next[(y) * TILE_X + (x)]

void lazy_init(void) {
  lazy_table_cur = malloc(TILE_X * TILE_Y * sizeof(unsigned));
  lazy_table_next = malloc(TILE_X * TILE_Y * sizeof(unsigned));
  memset(lazy_table_cur, 1, TILE_X * TILE_Y * sizeof(unsigned)); // Forcer le calcul initial
  memset(lazy_table_next, 0, TILE_X * TILE_Y * sizeof(unsigned));
}

void lazy_finalize(void) {
  free(lazy_table_cur);
  free(lazy_table_next);
}

void swap_tables_lazy(void) {
  unsigned *tmp = lazy_table_cur;
  lazy_table_cur = lazy_table_next;
  lazy_table_next = tmp;
  memset(lazy_table_next, 0, TILE_X * TILE_Y * sizeof(unsigned));
}

int tile_has_active_neighbor(int ty, int tx) {
  for (int dy = -1; dy <= 1; dy++) {
    for (int dx = -1; dx <= 1; dx++) {
      int ny = ty + dy;
      int nx = tx + dx;
      if (ny >= 0 && ny < TILE_Y && nx >= 0 && nx < TILE_X)
        if (lazy_cur(ny, nx)) return 1;
    }
  }
  return 0;
}

unsigned life_compute_omp_lazy(unsigned nb_iter) {
  unsigned res = 0;

  for (unsigned it = 1; it <= nb_iter; it++) {
    unsigned change = 0;

    #pragma omp parallel for collapse(2) reduction(|:change)
    for (int ty = 0; ty < TILE_Y; ty++) {
      for (int tx = 0; tx < TILE_X; tx++) {
        if (tile_has_active_neighbor(ty, tx)) {
          int x = tx * TILE_W;
          int y = ty * TILE_H;
```

```
50            int local_change = do_tile(x, y, TILE_W, TILE_H);
51            lazy_next(ty, tx) = local_change;
52            change |= local_change;
53          }
54        }
55      }
56
57      swap_tables();
58      swap_tables_lazy();
59
60      if (!change) {
61        res = it;
62        break;
63      }
64    }
65
66    return res;
67  }
```

Listing 1 – Code d'implémentation de la version Lazy combinée avec OpenMP

Analyse des performances Cette optimisation permet de réduire significativement la charge de travail en évitant de recalculer les tuiles qui restent stables au fil des itérations. Elle est particulièrement bénéfique pour les jeux de données où certaines régions restent stables sur plusieurs itérations, permettant ainsi de gagner du temps de calcul.

4.2 Empreinte Mémoire Réduite

Motivation Dans la version initiale du Jeu de la Vie, chaque cellule de la grille était représentée par un entier de 32 bits (`cell_t`), ce qui est largement surdimensionné pour stocker un état binaire (vivant ou mort). Cette représentation entraîne une consommation mémoire inutilement élevée, en particulier lorsque la taille de la grille devient importante. Dans le contexte des architectures GPU, où la mémoire globale est précieuse et souvent limitée, il est crucial d'optimiser l'empreinte mémoire pour maximiser la taille des grilles simulables et améliorer les performances globales.

Principe de l'optimisation L'optimisation consiste à utiliser un type de données plus adapté : un simple octet (`char`, soit 8 bits) par cellule. Ce choix constitue un compromis efficace : il permet de représenter l'état binaire de chaque cellule tout en divisant par quatre la mémoire utilisée par rapport à une représentation sur 32 bits. Cette réduction de l'empreinte mémoire permet :

- d'augmenter la taille maximale des grilles pouvant être traitées sur le GPU,
- de réduire la pression sur la bande passante mémoire.

Implémentation L'implémentation de cette optimisation a été réalisée dans le fichier `life_char.c`, qui reprend la structure du fichier `life.c` d'origine, mais en adaptant le type des cellules et les fonctions associées. Les principales étapes sont les suivantes :

1. **Création d'un nouveau fichier kernel** : Un nouveau fichier `life_char.c` a été créé pour isoler cette version et éviter les conflits lors de l'édition des liens (*linking*).
2. **Changement de type** : Toutes les occurrences du type `cell_t` ont été remplacées par `char`. Les allocations mémoire ont été mises à jour pour tenir compte de la nouvelle taille des cellules.
3. **Ajout du mot-clé `restrict`** : Pour optimiser davantage les accès mémoire, le mot-clé `restrict` a été ajouté à la déclaration des pointeurs vers les deux tables de cellules

(avant et après), permettant au compilateur d'optimiser les accès en supposant l'absence de recouvrement mémoire.

4. **Vérification de l'empreinte mémoire** : L'option `-debug u` a été utilisée lors de l'exécution du kernel pour observer précisément l'empreinte mémoire et valider la réduction attendue.

4.3 Version First Touch

Motivation et principe Sur les architectures multi-processeurs modernes, la mémoire est souvent organisée selon le modèle NUMA (*Non-Uniform Memory Access*), où chaque processeur (ou groupe de cœurs) dispose de sa propre mémoire locale, mais peut également accéder à la mémoire des autres nœuds avec une latence accrue. Par défaut, l'allocation mémoire peut ne pas tenir compte de cette topologie, ce qui peut entraîner des accès fréquents à la mémoire distante, fortement pénalisants pour les performances.

La stratégie dite de *First Touch* vise à exploiter au mieux la hiérarchie mémoire NUMA. Elle consiste à s'assurer que chaque portion de données est physiquement placée sur le nœud mémoire local au thread qui l'utilisera. En effet, sous Linux, la première écriture sur une page mémoire détermine sur quel nœud NUMA la page sera allouée physiquement. Il est donc crucial que chaque thread touche (écrive) en premier les données qu'il manipulera par la suite.

Implémentation Pour mettre en œuvre cette stratégie, nous avons créé un nouveau fichier kernel, `life_ft.c`, dans lequel la fonction d'initialisation `life_ft_init` applique explicitement la politique de First Touch. L'allocation mémoire est réalisée à l'aide de `mmap` afin de contrôler précisément la distribution des pages mémoire. Ensuite, une boucle parallèle OpenMP, avec une répartition statique des itérations, écrit un octet sur chaque page de la grille principale et de la grille secondaire. Ce schéma garantit que chaque page mémoire est initialisée (touchée) par le thread qui la traitera lors des calculs, forçant ainsi son placement local sur le nœud NUMA du thread.

```
void life_ft_init(void)
{
  if (_table == NULL) {
    const unsigned size = DIM * DIM * sizeof(cell_t);

    PRINT_DEBUG('u', "Memory␣footprint␣=␣2␣x␣%d␣bytes\n", size);

    _table = mmap(NULL, size, PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    _alternate_table = mmap(NULL, size, PROT_READ | PROT_WRITE,
                            MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    // First touch : chaque thread touche les pages qu'il utilisera
    #pragma omp parallel for schedule(static)
    for (size_t i = 0; i < size; i += PAGE_SIZE) {
      ((char*)_table)[i] = 0;
      ((char*)_alternate_table)[i] = 0;
    }
  }
}
```

Listing 2 – Initialisation First Touch

Calcul parallèle avec First Touch La version de calcul parallèle `life_ft_compute_omp` est identique à la version OpenMP classique, mais elle bénéficie d'une meilleure localisation des données en mémoire grâce à l'initialisation préalable.

```
unsigned life_ft_compute_omp(unsigned nb_iter)
{
  unsigned res = 0;

  for (unsigned it = 1; it <= nb_iter; it++) {
    unsigned change = 0;

    #pragma omp parallel for collapse(2) reduction(|:change) schedule(
       runtime)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        change |= do_tile(x, y, TILE_W, TILE_H);

    swap_tables();

    if (!change) {
      res = it;
      break;
    }
  }

  return res;
}
```

Listing 3 – Calcul parallèle avec First Touch

5 Vérification de Correction du Code - Comparaison des Hashages

Afin de valider que les optimisations appliquées (parallélisation, lazy allocation, etc.) ne modifient pas le comportement fonctionnel du programme, une vérification sémantique a été effectuée à l'aide de l'option `-sh`.

Le principe est simple : l'option `sh` permet de générer un hash de l'état final de la grille (l'image résultante après toutes les itérations). Deux programmes sont considérés fonctionnellement équivalents si, pour un même jeu de données, ils produisent le même hash. Cela garantit que les optimisations respectent la logique du programme séquentiel original (`life_seq`).

Pour effectuer cette vérification, nous avons exécuté directement les commandes suivantes :

```
./run -k life --arg moultdiehard1398 -s 256 -sh -n
./run -k life --arg random -i 19 -s 256 -sh -n

diff data/hash/life-seq-default-dim-256-iter-19-arg-random.sha256 \
     data/hash/life-seq-default-dim-256-iter-1399-arg-moultdiehard1398.sha256
```

Ces commandes permettent de générer les fichiers de hash et de les comparer pour vérifier s'il y a des différences entre deux versions du kernel.

6 Implémentation GPU avec CUDA

6.1 Explication détaillée de l'implémentation

L'objectif de cette partie est de comparer l'efficacité du calcul parallèle sur GPU avec CUDA face à une version séquentielle CPU, pour l'opération d'addition de deux vecteurs de taille N.

Le code `kernel.cu` se compose des étapes suivantes :

- **Allocation et initialisation** : Deux vecteurs `h_A` et `h_B` sont alloués et remplis avec des valeurs aléatoires sur le CPU.
- **Référence CPU** : Une addition séquentielle des vecteurs est réalisée sur le CPU (`vec_add_cpu`) pour servir de référence.
- **Transfert des données** : Les vecteurs sont copiés de la mémoire hôte (CPU) vers la mémoire du GPU (device) via `cudaMemcpy`.
- **Lancement du kernel CUDA** : Le kernel `vec_add` est lancé avec un nombre de blocs et de threads adapté à la taille N. Chaque thread calcule la somme d'un couple d'éléments.
- **Vérification** : Le résultat est copié du GPU vers le CPU puis comparé à la version séquentielle pour valider le calcul.
- **Mesure des temps** : Les temps de transfert (host-to-device et device-to-host) et le temps d'exécution du kernel sont mesurés séparément à l'aide des événements CUDA (`cudaEventRecord`).
- **Variation de N** : L'expérience est répétée pour différentes tailles de vecteurs (N), de 2^{10} à 2^{25}, afin d'analyser l'évolution des performances (pour la génération du graphique de performance).

Détail du kernel CUDA :

```
__global__ void vec_add(const float *a, const float *b, float *c, int N)
    {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N) {
        c[i] = a[i] + b[i];
    }
}
```

Listing 4 – Kernel CUDA pour l'addition de deux vecteurs

Chaque thread s'occupe d'une position i dans les vecteurs, ce qui permet d'effectuer l'opération en parallèle sur tout le tableau.

6.2 Analyse détaillée des temps

Le graphique ci-dessous présente le temps d'exécution (échelle logarithmique) en fonction de la taille des vecteurs N, pour le CPU et le GPU.

Analyse :

- **Pour les petites tailles** ($N < 10^4$), le GPU n'est pas forcément plus rapide que le CPU. Cela vient du temps nécessaire pour transférer les données entre la mémoire du processeur (CPU) et celle de la carte graphique (GPU), ainsi que du temps pour lancer le programme sur le GPU. Quand les vecteurs sont petits, ces étapes prennent plus de temps que le calcul lui-même.
- **Pour les tailles moyennes** ($10^4 < N < 10^6$), le GPU devient plus intéressant. Le CPU met de plus en plus de temps à faire les calculs, alors que le GPU garde un temps presque constant, grâce au grand nombre de calculs qu'il peut faire en même temps.
- **Pour les grandes tailles** ($N > 10^6$), la différence est très nette. Le GPU est beaucoup plus rapide. À ce niveau, le temps pour copier les données devient moins important que le gain obtenu grâce à la vitesse du calcul parallèle.

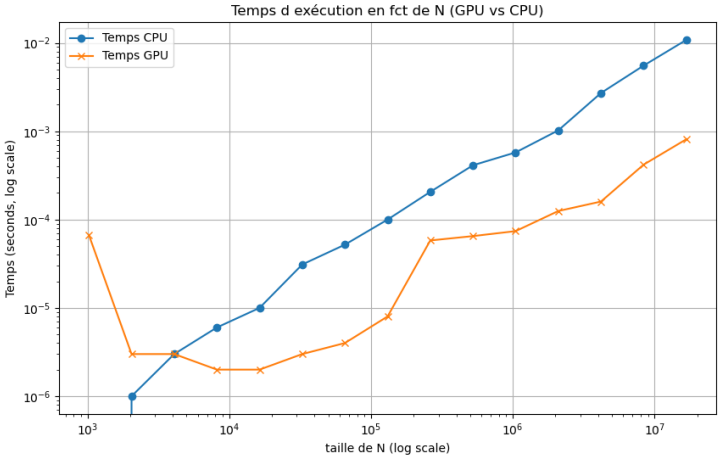


FIGURE 4 – Temps d'exécution en fonction de la taille de N (GPU vs CPU)

— **Comportement général du GPU** : On voit que le temps de calcul sur GPU augmente très peu quand on augmente la taille des vecteurs. Cela montre bien que le GPU peut faire beaucoup de calculs en même temps, ce qui le rend très efficace pour ce type d'opérations.

7 Analyse de performances

7.1 Analyse de résultat pour la version lazy (combiné avec omp) avec la version omp classique

Le graphe ci-dessous compare les performances de deux implémentations parallèles (omp et omp_lazy) du noyau "life" avec une grille de taille 4096, une tuile de taille 32 et un ordonnancement statique, sur trois jeux de données : moultdiehard130, random et ship. L'axe des abscisses représente le nombre de threads, l'axe des ordonnées le temps d'exécution moyen (ms).

7.1.1 Jeu de données : moultdiehard130

— L'implémentation `omp_lazy` est nettement plus rapide que `omp` pour tous les nombres de threads.
— Le temps d'exécution de `omp_lazy` chute rapidement dès les premiers threads et atteint un plateau très bas.
— La version `omp` diminue aussi mais reste significativement plus lente, surtout au-delà de 8 threads où la courbe stagne.

7.1.2 Jeu de données : random

— Les deux implémentations présentent des performances quasi identiques.
— Le temps d'exécution diminue fortement avec l'augmentation du nombre de threads, puis se stabilise.
— Aucune différence notable entre `omp` et `omp_lazy` sur ce jeu de données.

7.1.3 Jeu de données : ship

— `omp_lazy` est à nouveau beaucoup plus performant que `omp`.
— La différence est particulièrement marquée pour un petit nombre de threads.

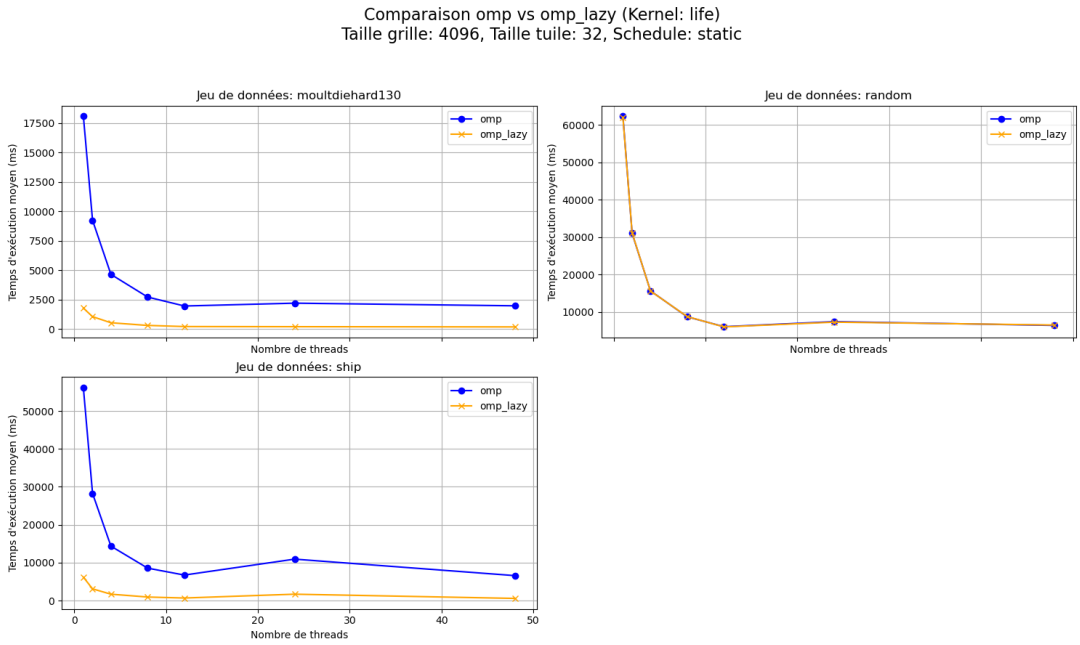


FIGURE 5 – Comparaison des temps d'exécution moyens (ms) entre `omp` et `omp_lazy` selon le nombre de threads, pour trois jeux de données.

— `omp_lazy` atteint rapidement un plateau de performance très bas, tandis que `omp` reste bien plus élevé.

7.1.4 Synthèse

— L'optimisation `omp_lazy` permet d'accélérer fortement les calculs sur les jeux de données *moultdiehard130* et *ship*, alors qu'elle n'apporte presque pas de différence sur *random*.
— Dans tous les cas, ajouter plus de threads améliore les temps d'exécution au début, mais au-delà d'un certain nombre, les gains deviennent très faibles.
— La façon dont les données sont organisées a donc un impact important sur l'efficacité de `omp_lazy` et sur les performances globales.

7.2 Analyse de l'empreinte mémoire des versions `life` et `life_char`

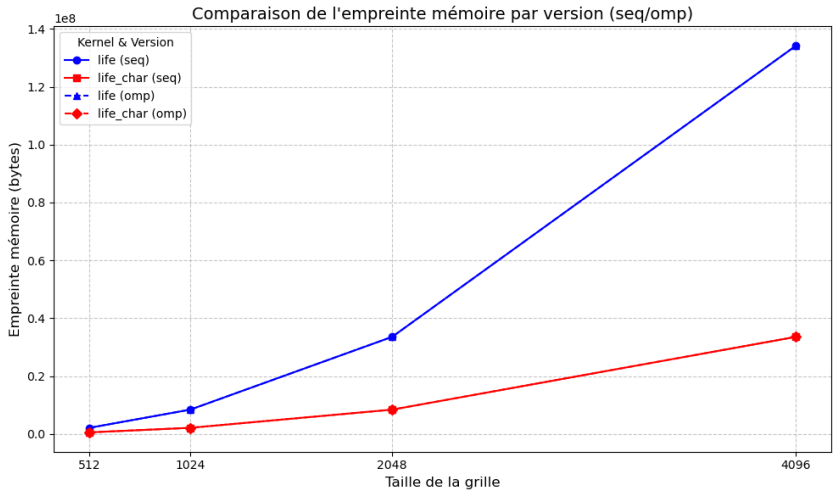


FIGURE 6 – Comparaison de l'empreinte mémoire entre les versions `life` et `life_char` (séquentiel et OMP) pour différentes tailles de grille.

Analyse approfondie

- Le graphe montre l'évolution de l'empreinte mémoire (en octets) en fonction de la taille de la grille pour deux versions du noyau : `life` et `life_char`, chacune déclinée en version séquentielle et parallèle (`omp`).
- On observe que, pour chaque taille de grille, la version `life_char` utilise beaucoup moins de mémoire que la version `life`, que ce soit en séquentiel ou en parallèle.
- Les courbes pour `life` séquentiel et `life` OMP sont confondues, tout comme celles pour `life_char` séquentiel et OMP. Cela signifie que l'utilisation de la mémoire ne dépend pas du mode d'exécution (séquentiel ou parallèle), mais uniquement de la structure de données utilisée (`life` ou `life_char`).
- Plus la taille de la grille augmente, plus la différence d'empreinte mémoire entre `life` et `life_char` devient importante. Pour la plus grande taille (4096), la version `life` dépasse 130 Mo, tandis que `life_char` reste sous 35 Mo.
- En résumé, le choix de la structure de données (`char` au lieu de `int` ou d'un type plus lourd) permet de réduire fortement la consommation mémoire, sans impact du mode séquentiel ou parallèle.

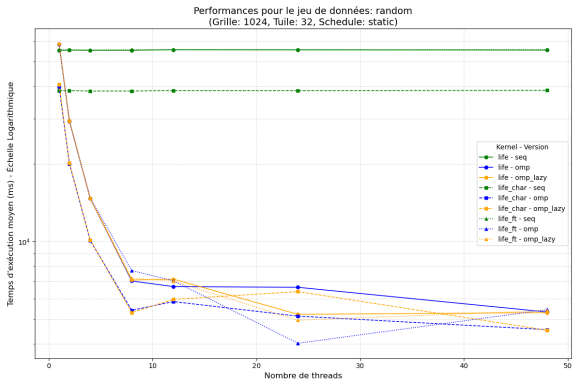
7.3 Analyse générale des différentes versions pour les différents jeux de données

Les trois figures suivantes présentent les temps d'exécution moyens (en millisecondes, échelle logarithmique) en fonction du nombre de threads pour différentes versions du kernel sur trois jeux de données : *random*, *ship* et *moultdiehard130*. Les tests ont été réalisés sur une grille de taille 1024, avec des tuiles de taille 32 et un ordonnancement (schedule) statique.

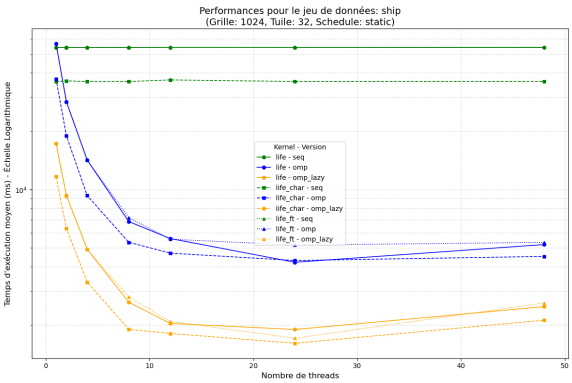
Comportement général

Les versions séquentielles (`seq`) affichent un temps d'exécution constant, indépendamment du nombre de threads, ce qui confirme l'absence de parallélisation.

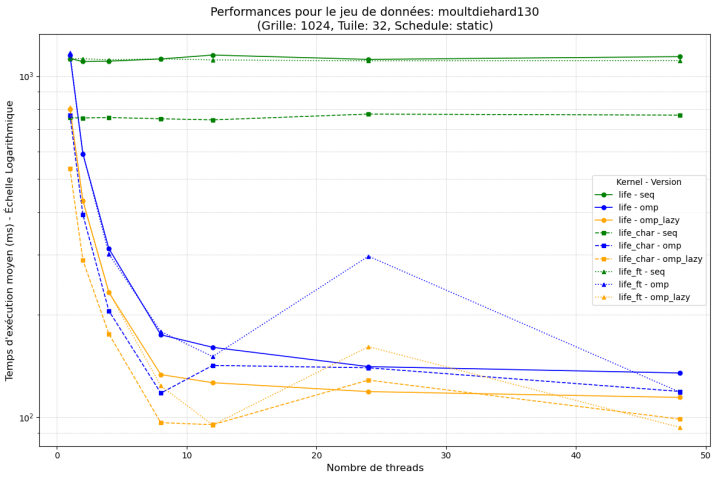
Les versions parallèles (`omp`, `omp_lazy`) montrent une forte diminution du temps d'exécution lorsque le nombre de threads augmente, puis un plateau, voire une légère remontée pour un nombre élevé de threads (au-delà de 24 à 32 threads).



(a) Jeu de données : random



(b) Jeu de données : ship



(c) Jeu de données : moultdiehard130

FIGURE 7 – Temps d'exécution moyen (ms) en fonction du nombre de threads pour trois jeux de données (échelle logarithmique).

Comparaison des versions

Les versions optimisées (`life_char` et `life_ft`) sont plus performantes que la version de base, que ce soit en séquentiel ou en parallèle.

L'approche `omp_lazy` surpasse la version OMP classique , particulièrement visible sur les jeux de données *ship* et *moultdiehard130*.

Impact du jeu de données

Le jeu *random* présente des temps d'exécution plus élevés que les deux autres, ce qui peut s'expliquer par une densité de cellules vivantes plus importante et donc un calcul plus coûteux.

Sur *ship* et *moultdiehard130*, les temps sont plus faibles et les courbes sont plus régulières, avec une meilleure scalabilité pour les versions optimisées.

8 Conclusion

Au cours de ce projet, nous avons étudié et optimisé le Jeu de la vie en partant d'une version séquentielle pour aller ensuite vers différentes stratégies d'optimisation. Nous avons d'abord mis en place une version OpenMP pour exploiter le parallélisme sur CPU, puis testé l'influence de

paramètres comme la taille des tuiles, le scheduling et le nombre de threads sur les performances. Ensuite, nous avons introduit des optimisations avancées, comme la version lazy (qui évite de recalculer les zones stables), la réduction de l'empreinte mémoire (en utilisant le type char au lieu d'un entier), et la stratégie First Touch pour mieux exploiter la mémoire sur les architectures NUMA. Les analyses ont montré que chaque optimisation apporte des gains différents selon le jeu de données et la configuration matérielle. La version lazy est très efficace sur certains jeux, la réduction mémoire permet de traiter des grilles plus grandes. En résumé, ce projet nous a permis de comprendre l'impact des différentes techniques de parallélisation et d'optimisation sur un problème concret, et d'illustrer l'importance de bien adapter ses choix aux caractéristiques des données et du matériel utilisé.

9 Annexes

Les annexes suivantes contiennent des informations complémentaires concernant la configuration matérielle utilisée pour les tests, ainsi que le script Bash développé pour automatiser la collecte des mesures de performance.

Annexe A : Configuration du Système Utilisé

Les tests ont été effectués sur une machine dotée des caractéristiques suivantes :

- **Processeur :** 24 unités de calculs logiques (12 cœurs physiques) avec une architecture NUMA. - **Mémoire :** 62 GB de RAM, répartis sur un seul nœud NUMA. - **Cache :** - L1d : 48KB (par cœur) - L1i : 32KB (par cœur) - L2 : 2MB à 4MB (par cœur) - L3 : 36MB partagé

La topologie de la machine est illustrée ci-dessous, avec la répartition des cœurs, des caches et des interconnexions PCI :

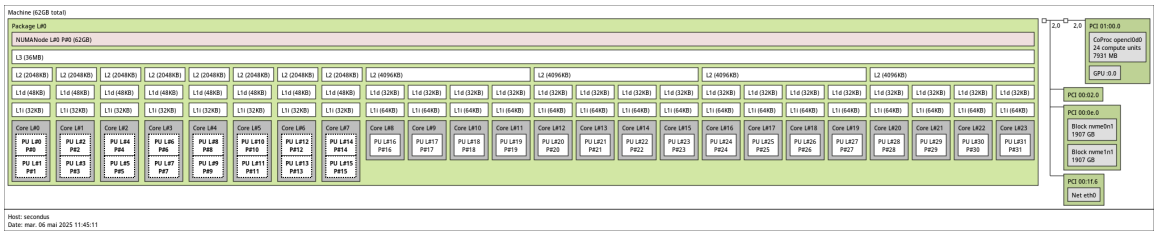


FIGURE 8 – Topologie matérielle de la machine utilisée (lstopo).

Annexe B : Configuration du GPU

La machine utilisée est équipée d'une carte graphique **NVIDIA GeForce RTX 4060**. Voici les caractéristiques principales :

- **CUDA Version :** 12.2 - **Driver NVIDIA :** 535.216.01 - **Mémoire GPU :** 8 GB GDDR6 - **Utilisation mémoire actuelle :** 411 MiB / 8188 MiB - **Puissance maximale :** 115W - **Température actuelle :** 31°C - **Processus actifs :** - Xorg : 252 MiB - xfwm4 : 3 MiB - Seed-version : 83 MiB - BrowserProcess : 62 MiB

Annexe C : Code Bash pour Automatiser la Prise des Mesures

Ce script Bash permet d'automatiser le processus de collecte des mesures de performance pour différentes configurations du kernel `life.c`. L'objectif est de générer un fichier CSV contenant les temps d'exécution pour divers paramètres, tels que la taille de grille, le nombre de threads, le type de scheduling et le kernel utilisé.

Fonctionnalités : - **Paramètres Interactifs :** Le script demande à l'utilisateur de saisir les paramètres suivants :

- Nom du fichier CSV de sortie
- Tailles des grilles
- Jeux de données (ex : `random`, `ship`)
- Tailles de tuiles (ex : 16, 32, 64)
- Nombre de threads
- Types de scheduling (`static`, `dynamic`, `guided`)
- Kernels (`life`, `life_char`, `life_ft`)
- Versions (`seq`, `omp`, `omp_lazy`)

- **Génération Automatique des Mesures :**

- Pour chaque combinaison de paramètres, le script exécute le kernel `life` avec les paramètres spécifiés et récupère le temps d'exécution.

— Les résultats sont stockés dans un fichier CSV avec les colonnes suivantes : `kernel, version, taille, jeu_donnees, tile, threads, schedule, iterations, temps_ms`

- Calcul Automatique des Itérations :

— Le nombre d'itérations est calculé dynamiquement en fonction de la taille de la grille pour garantir une charge de travail constante.

- Affichage des Informations :

— Le script affiche les informations relatives à chaque exécution pour permettre un suivi en temps réel.

```
#!/bin/bash

# ========================== CONFIGURATION ==========================

# Demander à l'utilisateur de saisir les paramètres
read -p "Nom du fichier CSV de sortie (ex: results/resultats_comp_versions_fixed.csv): " out
read -p "Tailles (séparées par un espace, ex: 512 1024 2048 4096): " -a tailles
read -p "Jeux de données (séparés par un espace, ex: random ship moultdiehard130): " -a donnees
read -p "Tailles de tuiles (séparées par un espace, ex: 16 32 64 128): " -a tuiles
read -p "Nombre de threads (séparés par un espace, ex: 1 2 4 8): " -a threads
read -p "Types de scheduling (séparés par un espace, ex: static dynamic guided): " -a schedules
read -p "Kernels (séparés par un espace, ex: life life_char life_ft): " -a kernel
read -p "Versions (séparées par un espace, ex: seq omp omp_lazy): " -a version

# Créer le fichier CSV et ajouter l'en-tête
echo "kernel,version,taille,jeu_donnees,tile,threads,schedule,iterations,temps_ms" > "$out"

# ===================================================================

for taille in "${tailles[@]}"; do
  iter=$(( (100 * 8192 * 8192) / (taille * taille) ))
  for jeu in "${donnees[@]}"; do
    for tile in "${tuiles[@]}"; do
      for th in "${threads[@]}"; do
        for ker in "${kernel[@]}"; do
          export OMP_NUM_THREADS=$th
          for sched in "${schedules[@]}"; do
            for ver in "${version[@]}"; do

              export OMP_SCHEDULE=$sched

              output=$(../run -k $ker -v $ver -a $jeu -s $taille -ts $tile -i $iter -n 2>&1)
              temps=$(echo "$output" | grep -Eo '^[0-9]+\.[0-9]+$' | tail -1)

              echo "$ker, $ver,$taille,$jeu,$tile,$th,$sched,$iter,$temps" >> "$out"
            done
          done
```

```
38          done
39        done
40      done
41    done
42  done
43
44  echo "Résultats␣générés␣dans␣$out"
```