Programmation des architectures parallèles - Rapport 1

Louis Baggio, Emilien Gallon

Fevrier 2024

université de BORDEAUX

Introduction

Ce rapport se divise en deux parties principales : la première partie est consacrée à l'optimisation de la fonction de calcul de tuiles et à l'implémentation des versions OpenMP du noyau `ssandPile`, à savoir `omp_for`, `omp_tiled` et `omp_taskloop`. La seconde partie détaille les expériences menées pour identifier les paramètres optimaux de chaque fonction.

Implementations et optimisations

Optimisation du calcul des tuiles

Le mot-clef **restrict** placé devant la variable `TABLE` informe le compilateur que les accès à ce tableau se feront exclusivement via ce pointeur, sans interférence d'autres pointeurs ou références au même tableau. Cela permet au compilateur d'optimiser les accès mémoire et d'améliorer la performance du code.

```
static TYPE *restrict TABLE = NULL;

int ssandPile_do_tile_opt(int x, int y, int width, int height)
{
  int diff = 0; // Variable to track whether any cells topple

  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
    {
      TYPE t_out_ij = table(in, i, j) % 4;
      t_out_ij += table(in, i + 1, j) / 4;
      t_out_ij += table(in, i - 1, j) / 4;
      t_out_ij += table(in, i, j + 1) / 4;
      t_out_ij += table(in, i, j - 1) / 4;
      table(out, i, j) = t_out_ij;
      diff |= t_out_ij >= 4; // Set 'diff' to 1 to indicate that there was a topple
    }

  return diff;
}
```

Plusieurs optimisations ont été réalisées sur `do_tile` :

- Utilisation d'une Variable Temporaire : Une variable temporaire `t_out_ij` est utilisée pour calculer la nouvelle valeur de chaque cellule avant de l'affecter à `table(out, i, j)`. Cela réduit le nombre d'accès à la mémoire pour lire et écrire dans table(out, i, j), puisqu'elle est mise à jour une seule fois par itération de boucle, après que tous les calculs sont terminés.
- Utilisation d'un **OU** logique : Remplacer une décision basée sur un branchement en une opération logique simple, qui peut être exécutée sans interruption du flux d'instructions, permet ainsi au processeur d'éviter les coûts associés à la prédiction de branchement. Cette approche exploite le fait que l'évaluation de `t_out_ij >= 4` produit un résultat booléen (0 ou 1 en C) qui est directement utilisable dans une opération de bit sans nécessiter un branchement.

Implementations OpenMP

For

```
unsigned ssandPile_compute_omp_for(unsigned nb_iter)
{
  unsigned it;
  int change = 1;

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

#pragma omp parallel for collapse(2) reduction(|:change) schedule(runtime)
    for (int i = 1; i < DIM - 1; i++)
    {
      for (int j = 1; j < DIM - 1; j++)
      {
        TYPE t_out_ij = table(in, i, j) % 4;
        t_out_ij += table(in, i + 1, j) / 4;
        t_out_ij += table(in, i - 1, j) / 4;
        t_out_ij += table(in, i, j + 1) / 4;
        t_out_ij += table(in, i, j - 1) / 4;
        table(out, i, j) = t_out_ij;
        change |= (t_out_ij >= 4);
      }
    }
    swap_tables();
  }

  return (change == 0) ? it - 1 : 0;
}
```

La directive OpenMP `pragma omp parallel for collapse(2) reduction(|:change)` est utilisée pour paralléliser les boucles imbriquées avec une réduction sur la variable change.

- `pragma omp parallel for` indique à OpenMP de distribuer les itérations de la boucle suivante entre les différents threads disponibles dans l'environnement d'exécution.
- `collapse(2)` indique à OpenMP de fusionner les deux niveaux de boucles imbriquées en une seule boucle traitant ainsi la double boucle comme une séquence linéaire d'itérations
- `reduction(|:change)` est utilisée pour combiner les valeurs de la variable `change` de tous les threads à la fin de la région parallèle. Chaque thread maintient sa propre copie locale de `change` pendant l'exécution de la boucle, et toutes ces copies locales sont combinées en utilisant le **OU** logique pour produire une seule valeur finale de `change`. Cette opération est thread-safe et garantit que les modifications concurrentes de `change` par plusieurs threads ne causeront pas de conflits de données.

Tiled

```
unsigned ssandPile_compute_omp_tiled(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int 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 + (x == 0), y + (y == 0),
                    TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                    TILE_H - ((y + TILE_H == DIM) + (y == 0)), omp_get_thread_num());
    swap_tables();
    if (!change)
      return it;
  }
  return 0;
}
```

Cette implémentation utilise les même directives OpenMP que celle de `omp_for` à la différence majeure que cette version permet l'utilisation de tuiles dont on peut faire varier les dimensions.

Taskloop

```
unsigned ssandPile_compute_omp_taskloop(unsigned nb_iter)
{
  unsigned it;
  int change = 1;

#pragma omp parallel
  {
#pragma omp single
    for (it = 1; it <= nb_iter && change != 0; it++)
    {
      change = 0;
#pragma omp taskloop collapse(2) shared(change)
      for (int y = 0; y < DIM; y += TILE_H)
        for (int x = 0; x < DIM; x += TILE_W)
        {
#pragma omp atomic
          change |=
              do_tile(x + (x == 0), y + (y == 0),
                      TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                      TILE_H - ((y + TILE_H == DIM) + (y == 0)), omp_get_thread_num());
        }
      swap_tables();
      if (!change)
      {
        break;
      }
    }
  }
  return (change == 0) ? it : 0;
}
```

L'objectif est d'exploiter les boucles de tâches d'OpenMP.

- `pragma omp parallel` : Cette directive crée une région parallèle, initialisant l'ensemble des threads.
- `pragma omp single` : À l'intérieur de la région parallèle, cette directive assure que le bloc de code qui suit est exécuté par un seul thread, celui qui va créer les tâches.
- `pragma omp taskloop collapse(2) shared(change)` : Cette directive est utilisée pour distribuer automatiquement les itérations des deux boucles imbriquées en tant que tâches OpenMP. L'option `collapse(2)` combine les deux boucles en une seule grande boucle pour une distribution efficace des tâches. La clause `shared(change)` spécifie que la variable `change` est partagée entre les tâches, permettant à toutes les tâches de mettre à jour cette variable pour signaler un changement.
- `pragma omp atomic` : Avant de modifier la variable `change`, cette directive garantit que l'opération est effectuée atomiquement, c'est-à-dire sans interruption, pour éviter les conflits d'écriture entre les tâches. Cela est crucial car `change` est partagée et modifiée par plusieurs tâches pouvant s'exécuter simultanément.

Recherche des paramètres optimaux

Toutes les experiences ont ete realisees en SSH sur la machine **Chagall** de la salle 008 du cremi.

Pour chaque taille de problème le nombre d'itérations a été choisi de façon à obtenir des temps d'exécutions proches de 500 millisecondes.

Le nombre de runs est fixé à 5 pour minimiser les artefacts.

La version optimisée de `do_tile` est utilisée systématiquement, y compris pour l'exécution de la version `seq` sur laquelle est calculé le speedup.

Recherche du nombre threads optimal

Dans le cadre de la recherche du nombre de threads optimal la politique de scheduling utilisée est `SCHEDULE=static` et la taille des tuiles est celle par défaut à savoir `tile_size=32`.

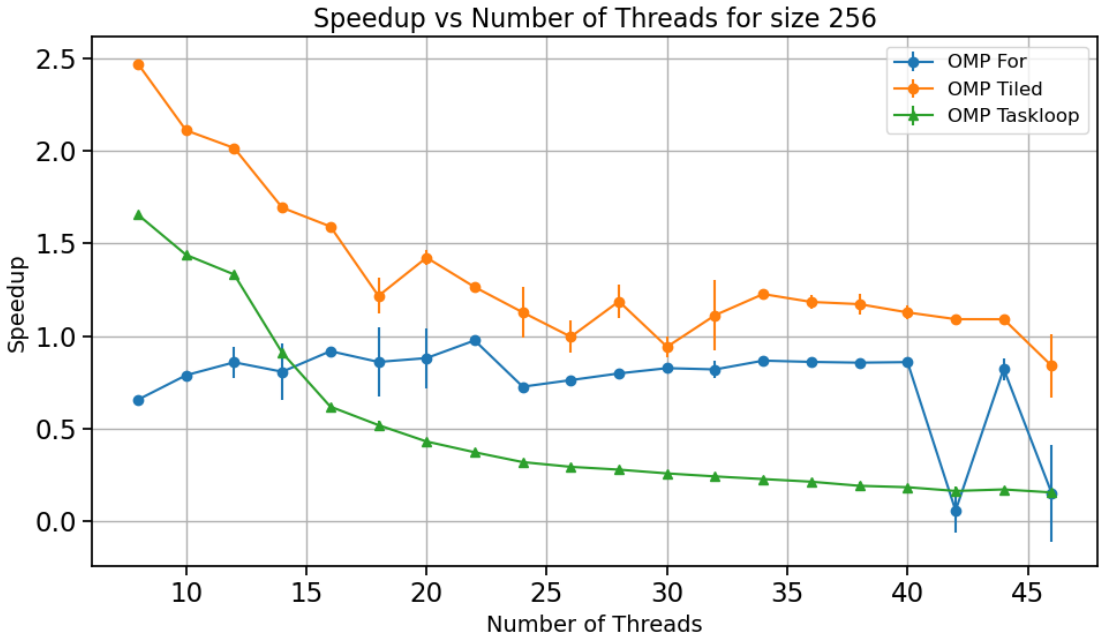


Figure 1: iterations=17035

Pour une taille de problème 256 on observe que la performance se dégrade rapidement lorsque le nombre de threads augmente. Le coût de création et de gestion des threads plus important combiné à une charge de travail légère tire la performance vers le bas car il n'y a pas assez de travail par thread pour compenser les coûts.

L'accélération est somme toute faible mais c'est attendu pour une petite taille de problème.

Le nombre de threads idéal trouvé est de 8 est la meilleure version est `omp_tiled`.

Pour la taille de problème 512 on retrouve cette dégradation de performance avec l'augmentation du nombre de threads mais de façon moins marquée. On note :

- Pic de performance de `omp_tiled` a 24 threads ce qui correspond au nombre de cœurs physiques de la machine.
- Instabilite de `omp_tiled` entre 26 et 38 threads
- Instabilité de `omp_for` pour un grand nombre de threads

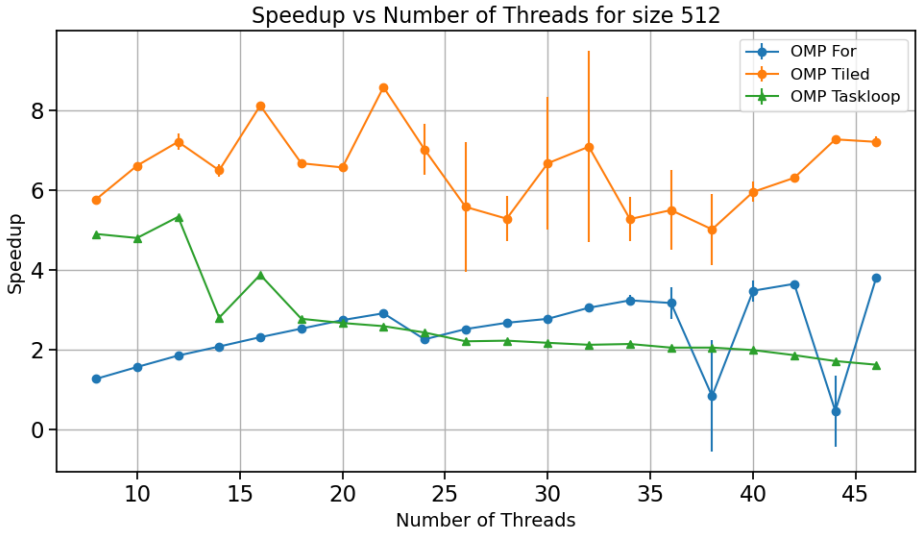


Figure 2: iterations=4096

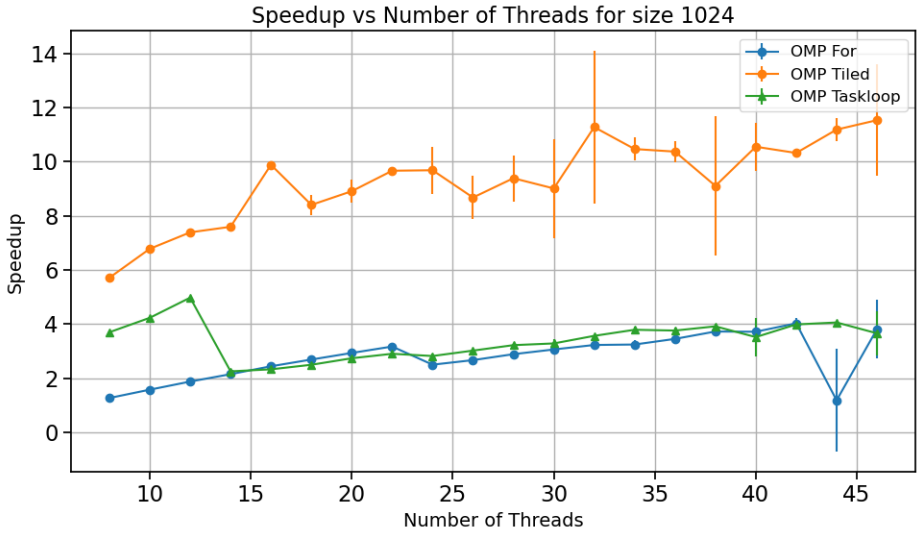


Figure 3: iterations=1024

Pour la taille du problème 1024 on observe cette fois une amélioration générale des performances avec l'augmentation du nombre de threads, cela signifie qu'on a atteint un seuil de charge de travail permettant de mobiliser toutes les ressources.

La version `omp_tiled` domine encore une fois les autres avec un pic notable a 32 threads (il y a du bruit sur le graphique mais des expériences plus précises de comparaison ont montré que 32 threads a bien de meilleures performances que 44)

La version `omp_taskloop` est plus performante pour 12 threads.

On note aussi que c'est pour cette taille de problème que le gain de performance des versions parallèles est le plus important.

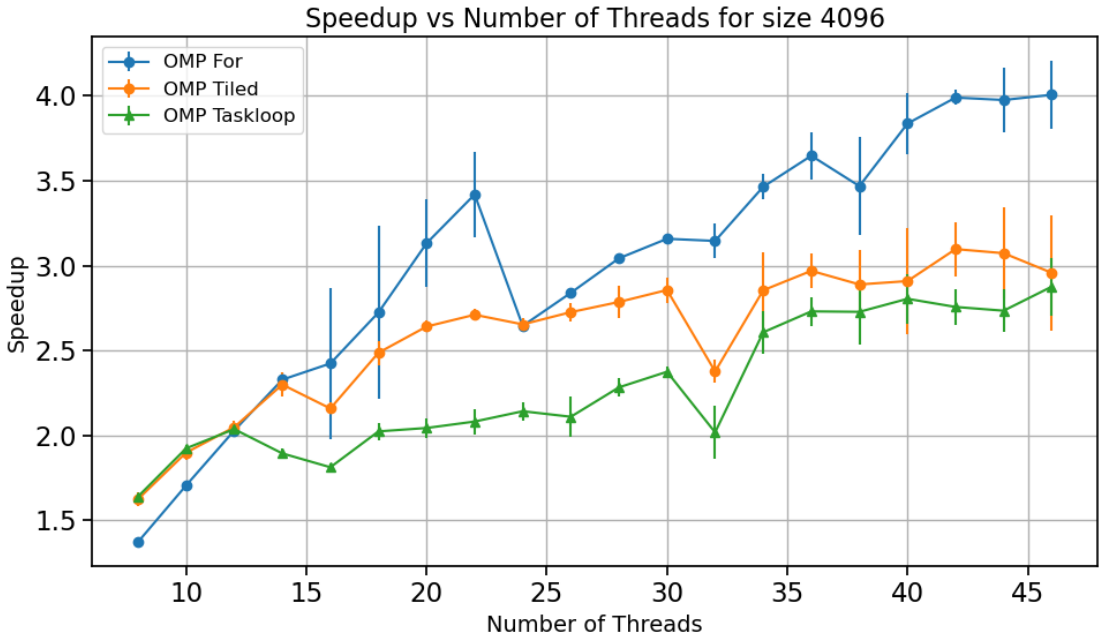


Figure 4: iterations=64

Pour la taille de probleme 4096 chaque version atteint un pic de performance pour un grand nombre de threads (44 ou 46)

Etonnamment c'est `omp_for` qui se détache du reste, l'explication est que l'on utilise une taille de tuile par défaut et donc on pénalise les versions tuilées.

A l'avenir on veillera à chercher à optimiser la géométrie des tuiles avant de faire la recherche du meilleur nombre de threads et de comparer les versions entre elles.

Dans la section suivante on pourra voir sur la carte de chaleur un speedup de plus de 4.5 pour `omp_tiled` avec une bonne géométrie de tuile, speedup supérieur à celui de `omp_for` (4).

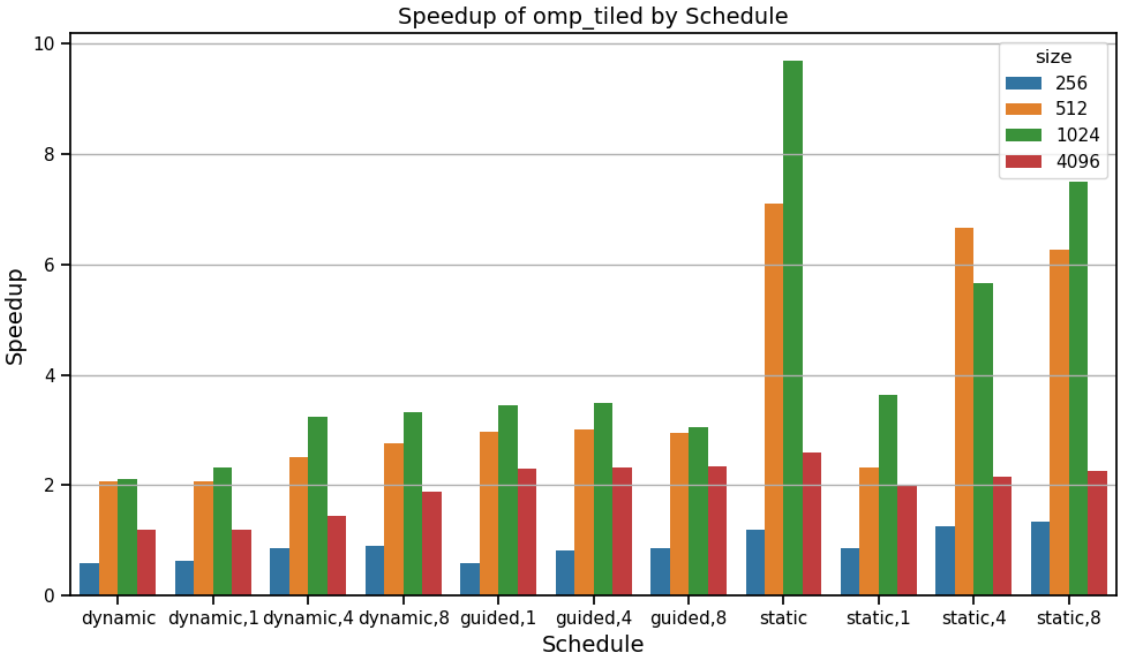
Recherche de la politique `schedule` optimale

Dans le cadre de la recherche de la politique de scheduling optimale le nombre de threads est fixé à `OMP_NUM_THREADS=24`, ce nombre n'est pas toujours l'optimal mais les deux versions testées y sont relativement égales en termes de performance.

Pour chaque taille de problème le nombre d'itérations est fixé comme auparavant :

- `size=256 -> pas de limite`
- `size=512 -> iterations=4096`
- `size=1024 -> iterations=1024`
- `size=4096 -> iterations=64`

Erratum: Pour une raison inconnue la version `omp_for` ne fonctionne qu'avec la politique static, les expériences suivantes ont donc été réalisées uniquement sur `omp_tiled`.



Le graphique montre que l'ordonnancement `static` est le plus performant, offrant la plus grande accélération pour toutes les tailles de problème. Cela suggère que la répartition du travail entre les threads est optimale, probablement en raison de la nature prévisible et uniformément répartissable de la charge de travail.

Les stratégies d'ordonnancement `dynamic` et `guided` ont des performances moins bonnes, cela pourrait indiquer que les coûts supplémentaires associés à la détermination dynamique de la répartition des tâches ne sont pas justifiés dans le cas de `omp_tiled`.

Recherche des valeurs `tile_height` et `tile_width` optimales

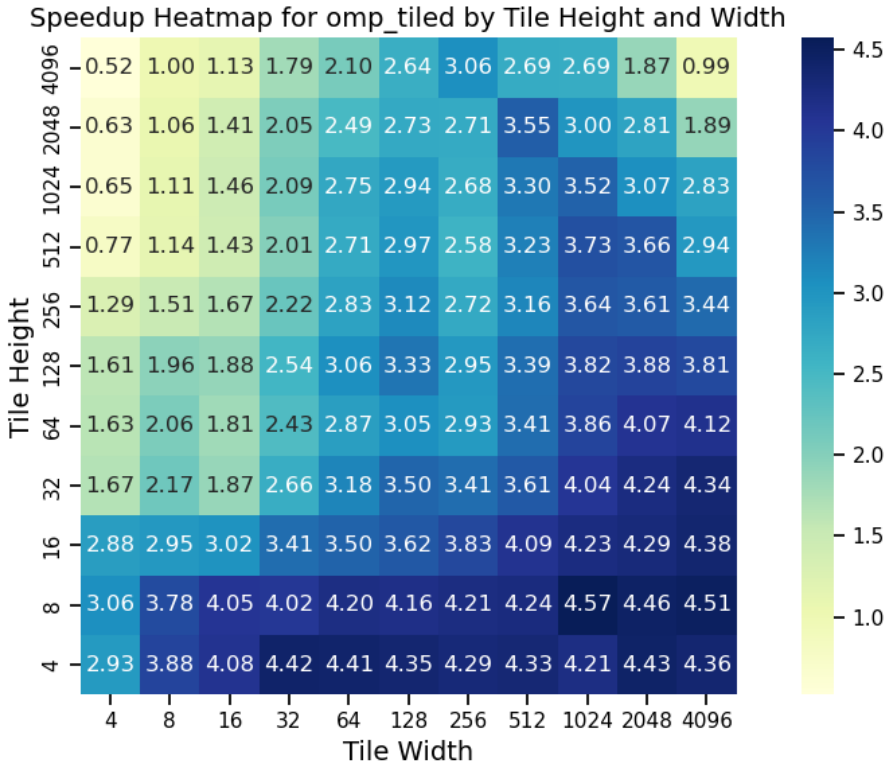
Dans le cadre de la recherche de la taille optimale des tuiles, la taille du problème est fixée à `size=4096` et le nombre d'itérations à `iterations=64`.

Un problème de grande taille est idéal car il permet de tester de plus grandes tailles de tuiles.

Le speedup est calculé par rapport à la version seq avec une taille de tuile par défaut `tile_size=32`.

On utilise la politique de scheduling `static` apres avoir constaté son efficacité plus haut.

Tiled

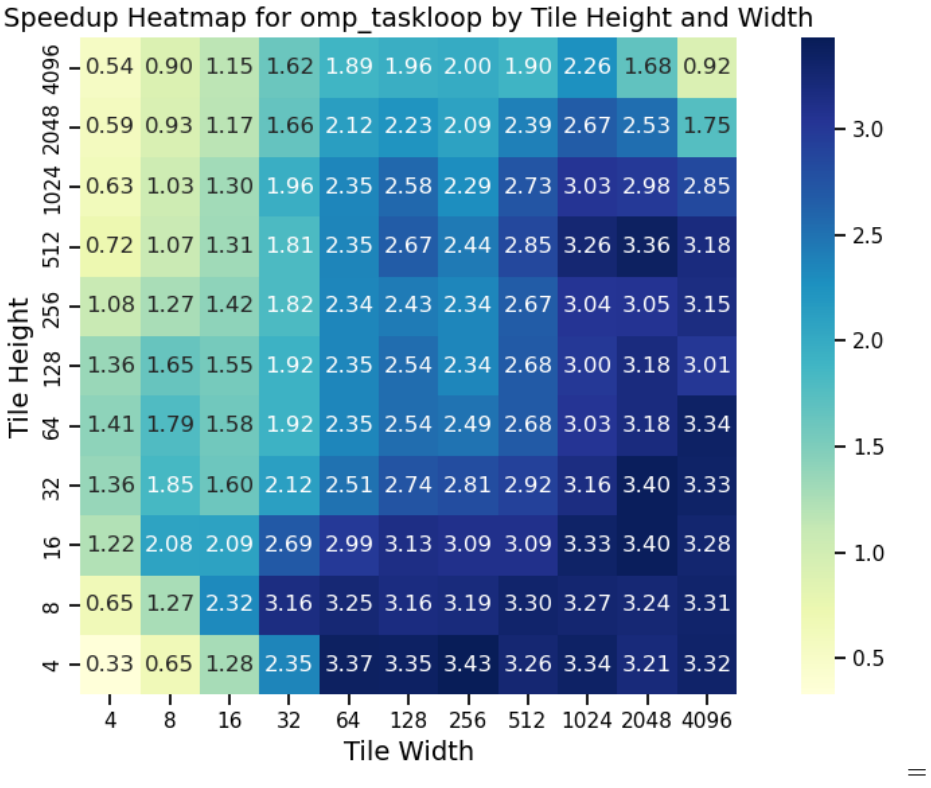


La meilleure accélération est observée avec des tuiles de largeur importante mais de faible hauteur.

Cette performance s'explique par le fonctionnement de la mémoire cache et la méthode de stockage des données. Lorsqu'un processeur accède à une variable, il charge également les données adjacentes dans la même zone de mémoire, qui correspondent généralement à la continuation de la ligne.

Ainsi, l'utilisation de tuiles larges favorise un traitement continu des données, optimisant l'accès à la mémoire. En revanche, l'utilisation de tuiles hautes nécessite un accès en colonne, contraignant le processeur à charger de nouvelles zones de mémoire pour chaque élément, ce qui ralenti le traitement.

Taskloop



=

La même observation peut être faite pour la version version `omp_taskloop`.

On notera cependant que le cette version semble moins bénéficier de la géométrie des tuiles, le speedup maximal étant de 3,43 pour `omp_taskloop` contre 4,57 pour `omp_tiled`. On peut imaginer que cela est dû à l'imprévisibilité de l'ordre d'exécution des tâches de `omp_taskloop`, ce qui peut réduire le bénéfice des optimisations d'accès à la mémoire.

Résultats

Mesure de l'optimisation de `do_tile`

L'optimisation de `do_tile` est efficace, le speedup est moindre sur la version séquentielle, de façon évidente l'exécution d'une fonction optimisée en parallèle permet des gains plus importants.

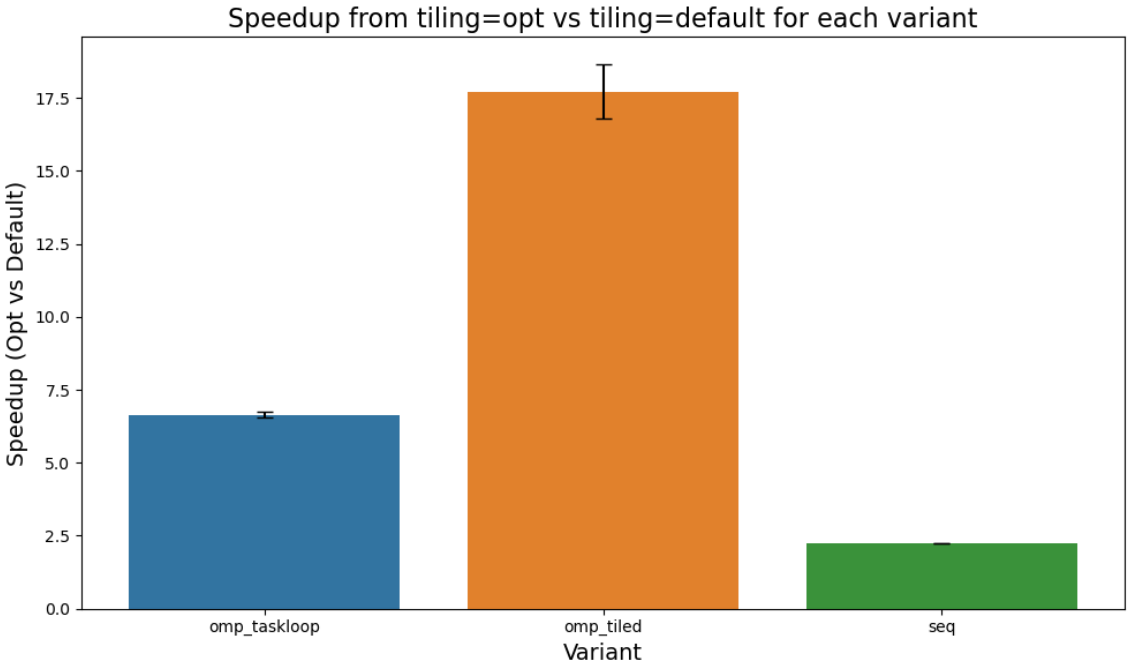


Figure 5: `threads=24 | schedule=static | size=4096 | iterations=64 | tile_size=32`

Speedup maximal

Le meilleur gain de performance possible est atteint sur le cas suivant :

- `size=1024`
- `iterations=1024`
- `threads=32`
- `schedule=static`
- `tile_height=16`
- `tile_width=256`

Variant	Mean Time (ms)	Error Margin (ms)	Speedup
seq	2,617.19	5.18	-
omp_tiled	79.24	8.75	**33.03**

Table 1: Speedup of `omp_tiled` vs `seq`