Comment faire de jolis pâtés

Aguibou Barry Lucas Palacz

9 avril 2022

1 Optimisations du pipeline

1.1 Avec le noyau synchrone

```
1  int ssandPile_do_tile_default(int x, int y, int width, int height) {
2      int diff = 0;
3
4      for (int i = y; i < y + height; i++)
5          for (int j = x; j < x + width; j++) {
6              table(out, i, j) = table(in, i, j) % 4;
7              table(out, i, j) += table(in, i + 1, j) / 4;
8              table(out, i, j) += table(in, i - 1, j) / 4;
9              table(out, i, j) += table(in, i, j + 1) / 4;
10             table(out, i, j) += table(in, i, j - 1) / 4;
11             if (table(out, i, j) >= 4)
12             diff = 1;
13         }
14
15     return diff;
16 }
```

Listing 1 – Code original de PA. Wacrenier

À la vue de ce code, la première idée que nous avons est de supprimer la série de `+=` sur les lignes 8 à 11, qui crée un effet de dépendance dans l'ordre des instructions. En remplacant par un humble `+` on permet au processeur d'éxécuter les opérations dans l'ordre qu'il l'arrange, voir de les faire en parallèle.

Dans un seconde temps, un camarade avec qui nous échangons sur le projet, Hugo Devidas, nous conseille d'ajouter l'attribut `restrict` au pointeur `TABLE` (qui contient la représentation des gains de sables). Cela a pour effet de diminuer considérablement le temps de calcul (voir graphique ci-dessous). En effet, nous pensons que l'usage de cet attribut nous évite beaucoup d'accès mémoire, dans lesquels on vérifie que la variable n'a pas été modifiée par concurrence entre deux calculs.

Finalement la dernière optimisation réalisée est celle des directives `#pragma` pour dérouler les boucles à la compilation et ainsi gagner légèrement en temps de calcul.

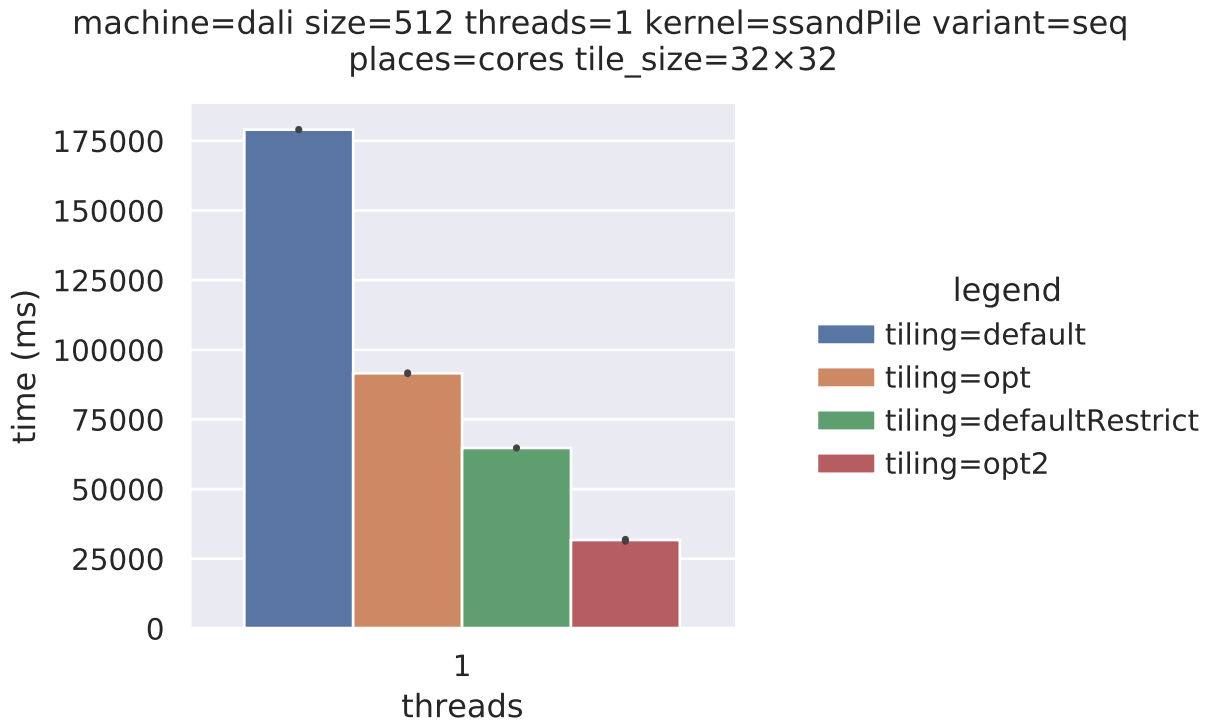

FIGURE 1 – Temps de calcul moyen (n=3) pour la version originale (default), la version originale mais TABLE est restrict (defaultRestrict), la première version optimisé (opt), la version optimisé finale, avec TABLE restrict et unroll-loops (opt2).

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
int ssandPile_do_tile_opt(int x, int y, int width, int height) {
    int diff = 0;
    for (int i = y; i < y + height; i++)
        for (int j = x; j < x + width; j++) {
            table(out, i, j) =
              table(in, i, j) % 4 +
              table(in, i+1, j) / 4 +
              table(in, i-1,j) / 4 +
              table(in, i, j+1) / 4 +
              table(in, i, j-1) / 4;

            if (table(out, i, j) >= 4)
                diff = 1;
        }

    return diff;
}
#pragma GCC pop_options
```

Listing 2 – Code optimisé final

1.2 Et pour le noyau asynchrone

```
int asandPile_do_tile_default(int x, int y, int width, int height) {
    int change = 0;
    for (int i = y; i < y + height; i++)
        for (int j = x; j < x + width; j++)
            if (atable(i, j) >= 4) {
                atable(i, j - 1) += atable(i, j) / 4;
                atable(i, j + 1) += atable(i, j) / 4;
                atable(i - 1, j) += atable(i, j) / 4;
                atable(i + 1, j) += atable(i, j) / 4;
                atable(i, j) %= 4;
                change = 1;
            }
    return change;
```

```
}
```

Listing 3 – Code original

La première idée que nous avons est de supprimer la partie de code répétitif `atable(i, j) / 4;` en utilisant une variable locale à la place. Cela a permis de gagner 10 % de performances. Nous avons aussi remarquer que, par l'utilisation de cette variable, nous ne sommes plus contraint d'avoir l'opération `atable(i, j) %= 4;` à la fin. En la placant au début nous remarquons un léger gain de vitesse aux alentours de 5 %. Nous pensons que cela peut-être lié au fait qu'on n'a optenu la valeur à l'initiation de la variable et que cette dernière se trouve maintenant dans un registre. Tandis que précédament nous devions aller la chercher plus loin dans les caches.

Nous avons ensuite rendu `TABLE restrict` et déroulé les boucles comme pour le noyau synchrone. Cela a égélement eu de bon résultats sur notre temps de calcul.

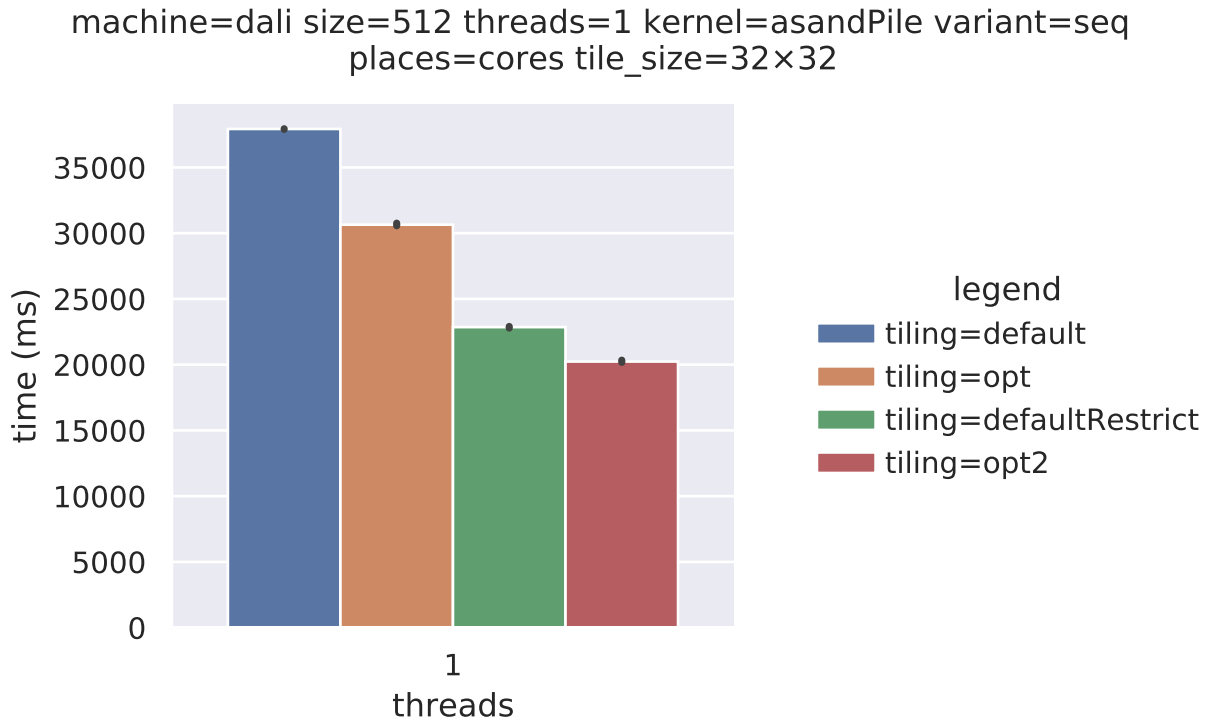

FIGURE 2 – Temps de calcul moyen (n=3) pour la version originale (default), la version originale mais TABLE est restrict (defaultRestrict), la première version optimisé (opt), la version optimisé finale, avec TABLE restrict et unroll-loops (opt2).

```c
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
int asandPile_do_tile_opt(int x, int y, int width, int height) {
    int change = 0;
    for (int i = y; i < y + height; i++)
        for (int j = x; j < x + width; j++)
            if (atable(i, j) >= 4) {
                TYPE distributedSand = atable(i, j) / 4;
                atable(i, j) %= 4;
                atable(i, j - 1) += distributedSand;
                atable(i, j + 1) += distributedSand;
                atable(i - 1, j) += distributedSand;
                atable(i + 1, j) += distributedSand;
                change = 1;
            }
    return change;
}
#pragma GCC pop_options
```

Listing 4 – Code optimisé final

2 Implémentation en AVX

2.1 Présentation du code

2.1.1 Version synchrone

Pour implémenter la version synchrone nous avons essayer de suivre le code de la version optimisé. Nous sommes initialement parti sur une première version la plus triviale possible, dans laquelle les bords de l'image sont calculé par la fonction opt non vectorisé [1]. Cependant, nous avons tout de même fait l'expérience de quelques désagrément de programmation qui nous a fait perdre de temps et refroidi à complexifié notre fonction en traitant les bords.
Cependant, une fois avoir compris comment se servir des fonctions intel pour AVX2(Listing 5) et AVX512(Listing 6), on a décider d'implémenter deux versions pour ce noyau, la première utilisant AVX2 avec des vecteurs de 8 et la seconde utilise AVX512 avec des vecteurs de 16. Pour ce faire nous avons de passage implémenter quelques fonctions intermédiaires à savoir **"__mm256__mod4__epi32(___m256i var)", "__mm256__div4__epi32(___m256i var)"** et **"__mm512__mod4__epi32(___m512i var)", "__mm512__div4__epi32(___m512i var)"** repectivement pour AVX2 et AVX512 pour le calcul de la division et le modulo et **"__mm256__testeq__si256(___m256i var1, ___m256i var1)"** pour faire le teste *"table(out, i, j) != table(in, i, j)"* à la fin de chaque traitement pour l'AVX2.

```
int ssandPile_do_tile_avx(int x, int y, int width, int height) {
    // First approach: does not treat border tiles
    if (x == 1 || x + width == DIM - 1 || y == 1 || y + height == DIM - 1)  {
        PRINT_DEBUG('k', "Tile (x,y)=(%d, %d) w:%d h:%d : treated as border\n", x, y,
                    width, height);
        return ssandPile_do_tile_opt(x, y, width, height);
    }

    int diff = 0;
// In a first time, we design this function to work on multiples of 8
//(one vector of ints 64bits)
// It should work on any square tiles with power of 2 width larger than 8x8.
// Exemple command: ./run -k ssandPile -v tiled -wt avx -s 256 -ts 64
    for (int i = y; i < y + height; i ++)
        for (int j = x; j < x + width; j += AVX_VEC_SIZE_INT) {
            __m256i original, center, left, right, bottom, top;
            original = center = _mm256_loadu_si256((__m256i *) &table(in, i, j));
            right = _mm256_loadu_si256((__m256i *) &table(in, i+1, j));
            left = _mm256_loadu_si256((__m256i *) &table(in, i-1, j));
            bottom = _mm256_loadu_si256((__m256i *) &table(in, i, j+1));
            top = _mm256_loadu_si256((__m256i *) &table(in, i, j-1));

            center = _mm256_mod4_epi32(center);
            right = _mm256_div4_epi32(right);
            left = _mm256_div4_epi32(left);
            bottom = _mm256_div4_epi32(bottom);
            top = _mm256_div4_epi32(top);

            __m256i res = _mm256_add_epi32 (center, right);
            res = _mm256_add_epi32(res, left);
            res = _mm256_add_epi32(res, bottom);
            res = _mm256_add_epi32(res, top);

            _mm256_storeu_si256((__m256i_u *) &table(out, i, j), res);

            diff |= !_mm256_testeq_si256(original, res);
        }
    return diff;
}
```

Listing 5 – Implémentation en avx du noyau synchrone avec AVX2

```
int ssandPile_do_tile_avx512(int x, int y, int width, int height) {
  if (x == 1 || x + width == DIM - 1 || y == 1 || y + height == DIM - 1) {
```

1. Nous avons pris soin de le vérifier en demandant les informations de vectorisation lors de la compilation. `opt` ne peut pas être vectorisé.

```
    PRINT_DEBUG('k', "Tile (x,y)=(%d, %d) w:%d h:%d : treated as border\n", x, y,
                width, height);
    return ssandPile_do_tile_opt(x, y, width, height);
  }

  int diff = 0;
  // Exemple command: ./run -k ssandPile -v tiled -wt avx512 -s 256 -ts 64
  for (int i = y; i < y + height; i ++)
    for (int j = x; j < x + width; j += AVX512_VEC_SIZE_INT) {
      __m512i original, center, left, right, bottom, top;
      original = center = _mm512_loadu_si512(&table(in, i, j));
      right = _mm512_loadu_si512(&table(in, i+1, j));
      left = _mm512_loadu_si512(&table(in, i-1, j));
      bottom = _mm512_loadu_si512(&table(in, i, j+1));
      top = _mm512_loadu_si512(&table(in, i, j-1));

      center = _mm512_mod4_epi32(center);
      right = _mm512_div4_epi32(right);
      left = _mm512_div4_epi32(left);
      bottom = _mm512_div4_epi32(bottom);
      top = _mm512_div4_epi32(top);

      __m512i res = _mm512_add_epi32 (center, right);
      res = _mm512_add_epi32(res, left);
      res = _mm512_add_epi32(res, bottom);
      res = _mm512_add_epi32(res, top);

      _mm512_storeu_si512(&table(out, i, j), res);

      diff |= _mm512_cmpgt_epi32_mask(res, original);
    }
  return diff;
}
```

Listing 6 – Implémentation en avx du noyau synchrone avec AVX512

2.1.2 Version asynchrone

Pour cette version asynchrone, nous avons encore écarté les bords du calcul, en effet `opt` ne nous a jamais trahis. Nous sommes cependant parti dès le début sur une version AVX512(Listing 7) afin de pouvoir utiliser la méthodes `_mm512_alignr_epi32()`.

```
int asandPile_do_tile_avx(int x, int y, int width, int height) {
    int diff = 0;
    if (x == 1 || x + width == DIM - 1 || y == 1 || y + height == DIM - 1)  {
        PRINT_DEBUG('k', "Tile (x,y)=(%d, %d) w:%d h:%d : treated as border\n",
        x, y, width, height);
        return asandPile_do_tile_opt(x, y, width, height);
    }
    __m512i zero = _mm512_setzero_epi32();
    __m512i three = _mm512_set1_epi32(3);

    for (int i = y; i < y + height; i ++)
        for (int j = x; j < x + width; j += AVX512_VEC_SIZE_INT) {
            __m512i center = _mm512_loadu_si512(&atable(i, j));
            __mmask16 unstable = _mm512_cmpgt_epi32_mask(center, three);

            if (unstable) {
                __m512i top, bottom;
                top = _mm512_loadu_si512(&atable(i - 1, j));
                bottom = _mm512_loadu_si512(&atable(i + 1, j));

                __m512i tmp1 = _mm512_srli_epi32(center, 2);
                center = _mm512_mod4_epi32(center);
                center = _mm512_add_epi32(center, _mm512_alignr_epi32(zero, tmp1, 1));
                center = _mm512_add_epi32(center, _mm512_alignr_epi32(tmp1, zero,
                            AVX512_VEC_SIZE_INT - 1));

```

```
                top = _mm512_add_epi32(top, tmp1);
                bottom = _mm512_add_epi32(bottom, tmp1);

                atable(i, j-1) += _mm256_extract_epi32(
                                        _mm512_extracti32x8_epi32(tmp1, 0), 0);
                atable(i, j+AVX512_VEC_SIZE_INT) += _mm256_extract_epi32(
                                                _mm512_extracti32x8_epi32(tmp1, 1), 7);

                _mm512_storeu_si512(&atable(i-1, j), top);
                _mm512_storeu_si512(&atable(i, j), center);
                _mm512_storeu_si512(&atable(i+1, j), bottom);

                diff = 1;
            }
        }
    return diff;
}
```

Listing 7 – Implémentation du noyau asynchrone en AVX512

2.2 Présentation des expériences

2.2.1 Noyau synchrone

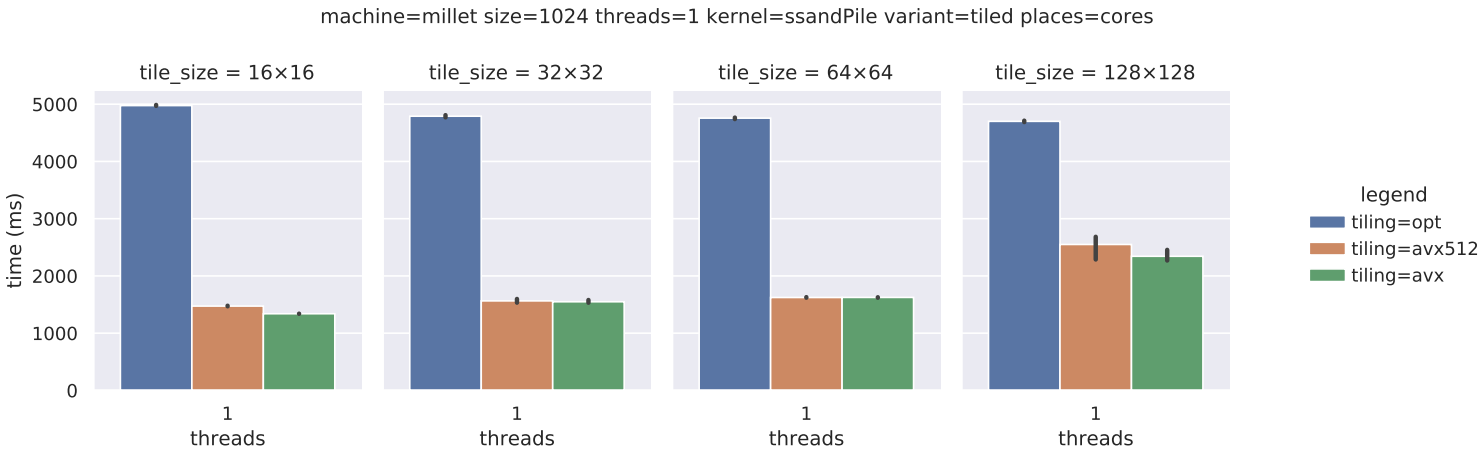

FIGURE 3 – Temps de calcul moyen (n=3) pour la version optimisé non-vectoriel (opt), la version AVX2 (avx) et AVX512 (avx512) pour 1000 itérations sur des tailles de tuiles variables. Variante `tiled` utilisé pour ne pas introduire de différence lié aux tuiles stables.

Lorsque les tuiles sont trop grandes, on voit que le temps de la version AVX se rapproche de la version non-optimisé. Logique puis-ce que notre implémentions ne traite pas les bords. Sur des dimensions de 1024 par 1024 et avec des tuiles de 128 par 128, 28 tuiles sont traités de façon non-vectoriel et 36 exploitent l'AVX.

Il est cependant intéressent de noter que les versions 16×16, 32×32 et 64×64 ont un peu près le même temps de calcul. Ce qui signifie que ce qui est perdu par l'appel plus récurent de la version non-vectoriel est compensé quelque part.

Nous avons essayé de retravailler pour ce rendu sur la fonction `ssandPile_compute_omp_lazy()`, qui était auparavant très **très** lente. Nous avons introduit de l'inter-dépendances entre les threads pour obtenir un bon résultat. Cependant le temps d'exécution de la fonction étant encore un peu trop lente, nous avons décidé de pousser nos tests du côté de la version asynchrone qui possède une version paresseuse bien mieux implémenté.

2.2.2 Noyau asynchrone

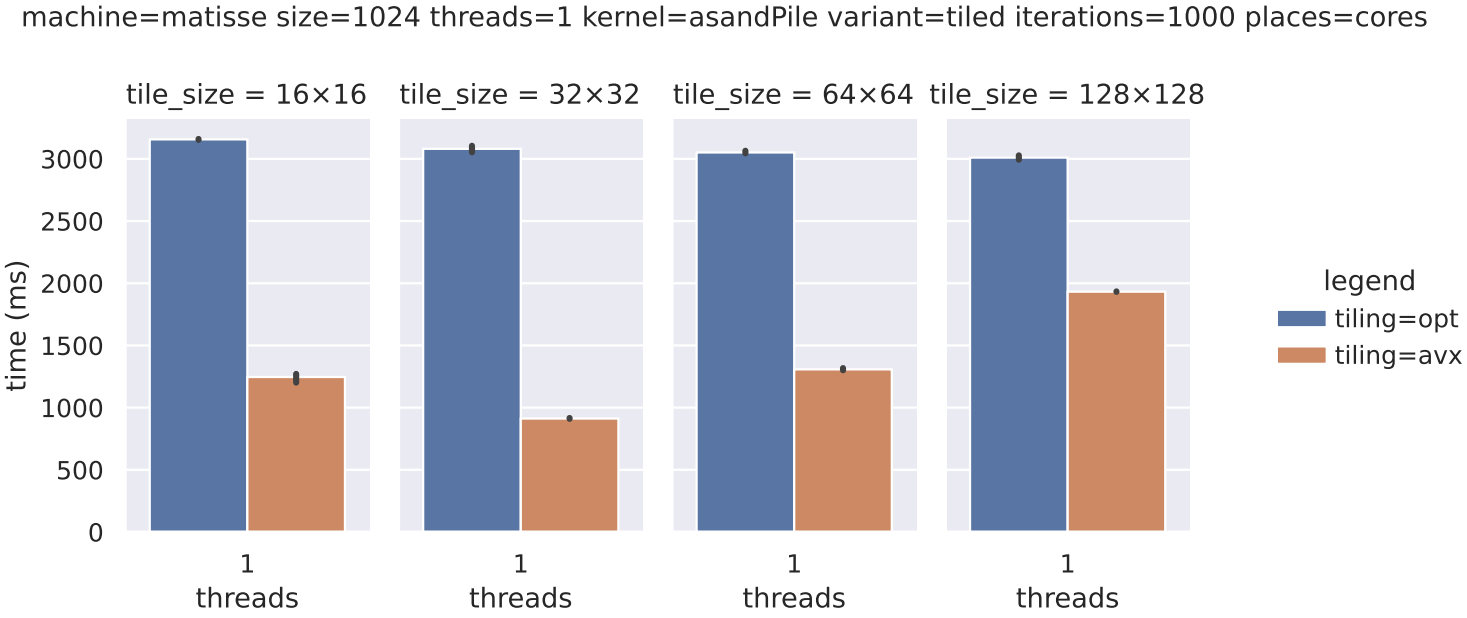

FIGURE 4 – Temps de calcul moyen (n=3) pour la version optimisé non-vectoriel (opt) et la version AVX512 (avx)pour 1000 itérations sur des tailles de tuiles variables. Variante `tiled` utilisé pour ne pas introduire de différence lié aux tuiles stables

— **omp_tiled** :
Suite au précedent rendu, nous avons constater que les meilleurs distributions pour cette implémentation sont `schedule static` et `schedule dynamic,4`, nous avons donc décider de faire des expériences sur ces distributions plus la distribution `schedule nonmonotonic:dynamic` pour cette implémentation. Ci-dessous des graphiques présentant la meilleure répartition des tuiles pour chaque politique de distribution sur une image(512) avec un nombre d'itérations que nous avons limité à 1000. Le `schedule static` conserve un *speedup* maximal parmi les trois distributions.

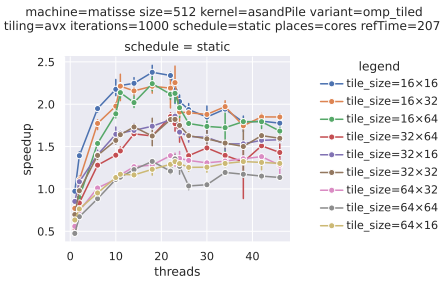

FIGURE 5 – Expérience avec `schedule static` pour différentes tailles de tuiles

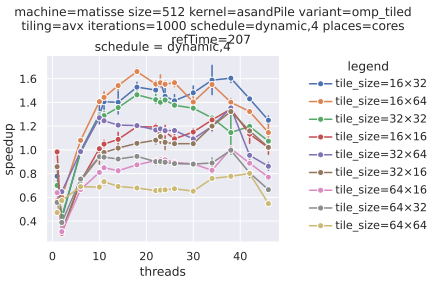

FIGURE 6 – Expérience avec `schedule dynami,4` pour différentes tailles de tuiles

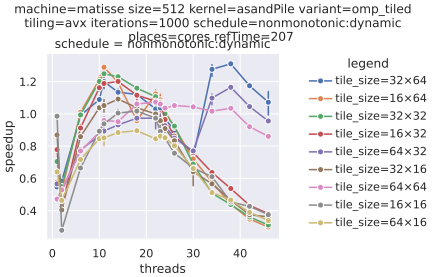

FIGURE 7 – Expérience avec `schedule nonmonotonic:dynamic` pour différentes tailles de tuiles

Dans l'optique de déterminer la meilleure répartition de tuiles, nous avons décider de faire une carte de chaleur sur différents points présentant un meilleur speedup (de 11 à 24 threads) sur la base de la précedente expérience sur la distribution statique et il savère que 15, 16 et 20 threads disposent le meilleur speedup voir(Figure 8).

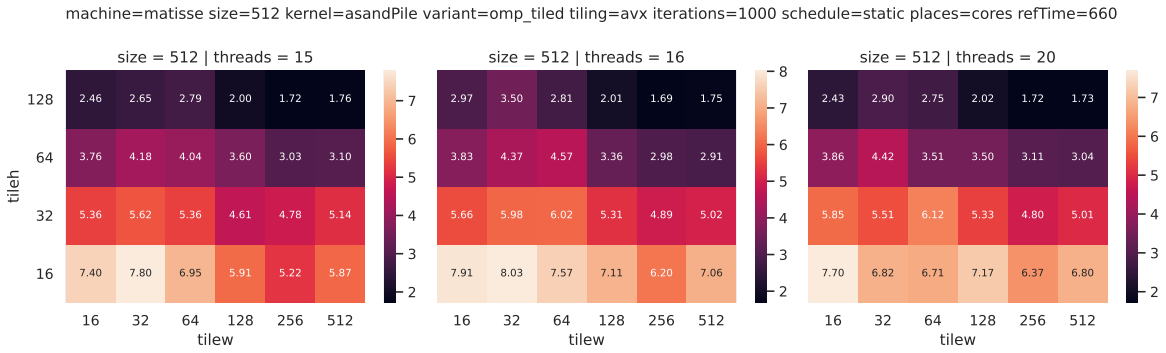

FIGURE 8 – Carte de chaleur du speedup moyen (n=3) pour la version `omp_tiled()` en fonction de la taille des tuiles au voisinage du nombre de thread aux performances maximales.

Sur la carte de chaleur ci-dessus, nous constatons que la répartition 16x32 avec 16 threads présente le meilleur speedup donc à présent nous allons faire une dernière expérience sans limiter le nombre d'itérations pour cette répartition dans le but de déterminer le speedup maximal pouvant être atteinte par ladite répartition.

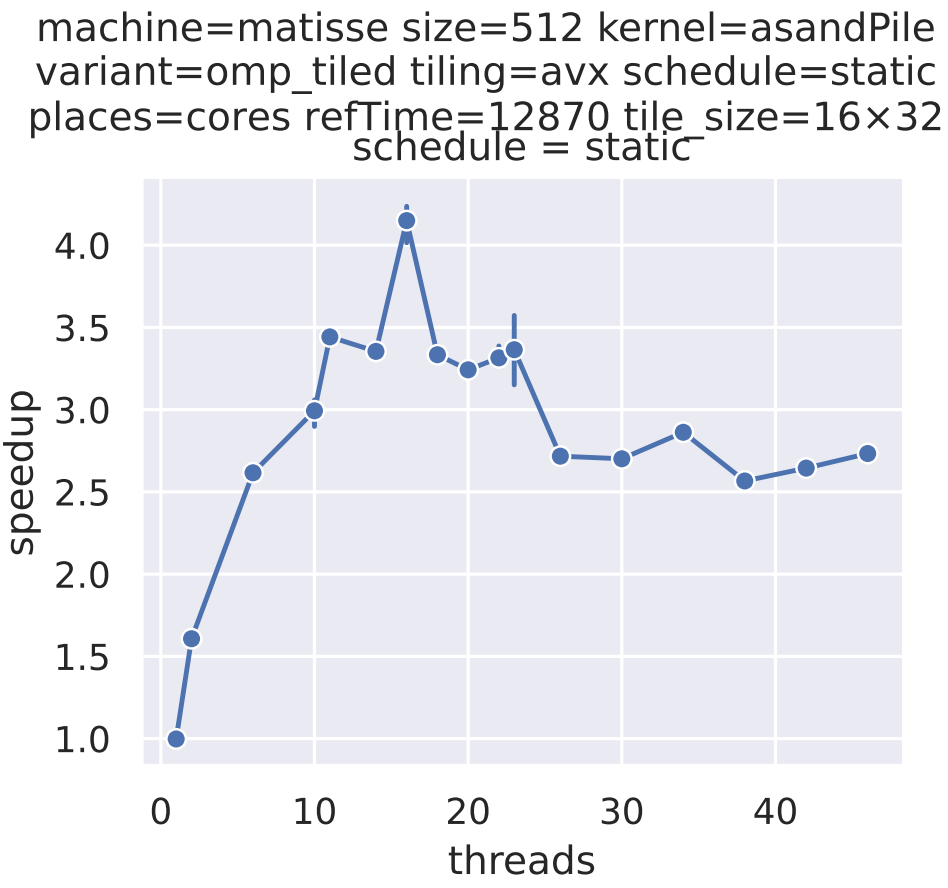

FIGURE 9 – Expérience complète avec `schedule static` pour tileh=16 et tilew=32

— **omp_lazy** : Vous remarquez peut-être que nous effectuons une comparaison sur 1000 itérations. Hors la méthode d'implémentation de la fonction `do_tile()` peut avoir des conséquences sur le nombre d'itérations contrairement à la version synchrone. Nous reviendrons là-dessus un peu plus tard.
A nouveaux on remarque ce gain de performance sur des tuiles un peu plus grandes, alors qu'on perd en vectorisation.

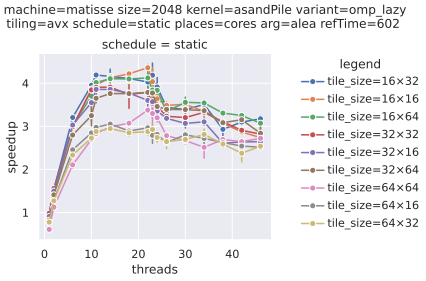

FIGURE 10 – Calcul du speedup moyen (n=3) en fonction du nombre de thread avec `schedule static`

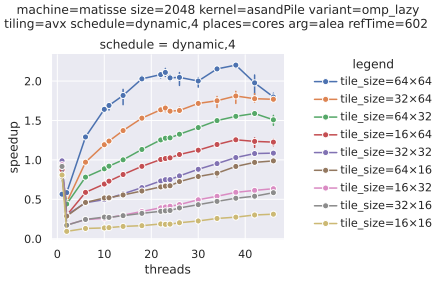

FIGURE 11 – Calcul du speedup moyen (n=3) en fonction du nombre de thread avec `schedule dynamic`

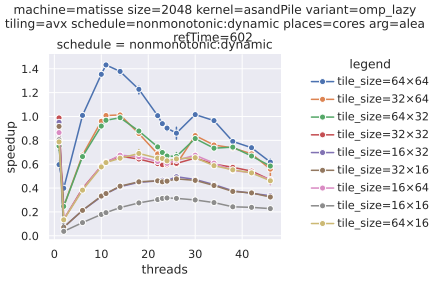

FIGURE 12 – Calcul du speedup moyen (n=3) en fonction du nombre de thread avec `schedule nonmonotonic:dynamic`

On tente à présent à savoir pour qu'elle paramètres `omp_lazy()` est le plus efficace. La version statique à le meilleur speedup. Si vous n'arrivez pas bien à voir pour qu'elle taille de tuile, ne vous en faites pas, nous vous avons fait une carte de chaleur, pour les nombres de threads les plus intéressants.

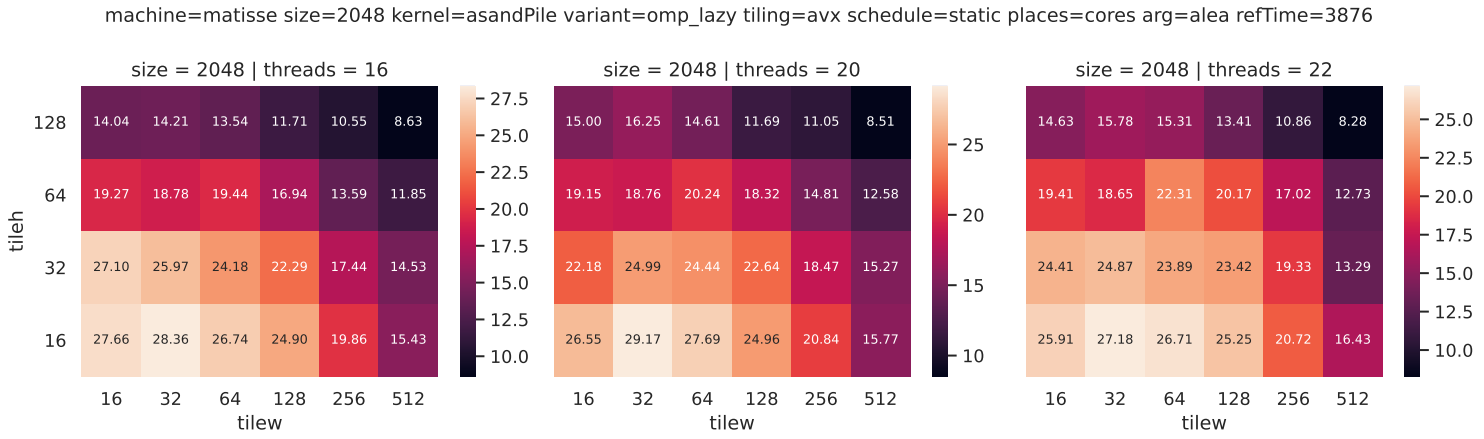

FIGURE 13 – Carte de chaleur du speedup moyen (n=3) pour la version `omp_lazy()` en fonction de la taille des tuiles au voisinage du nombre de thread aux performances maximales.

Effectuons à présent une comparaison du meilleur paramétrages vectoriel et non-vectoriel pour observer la granularité des tâches.

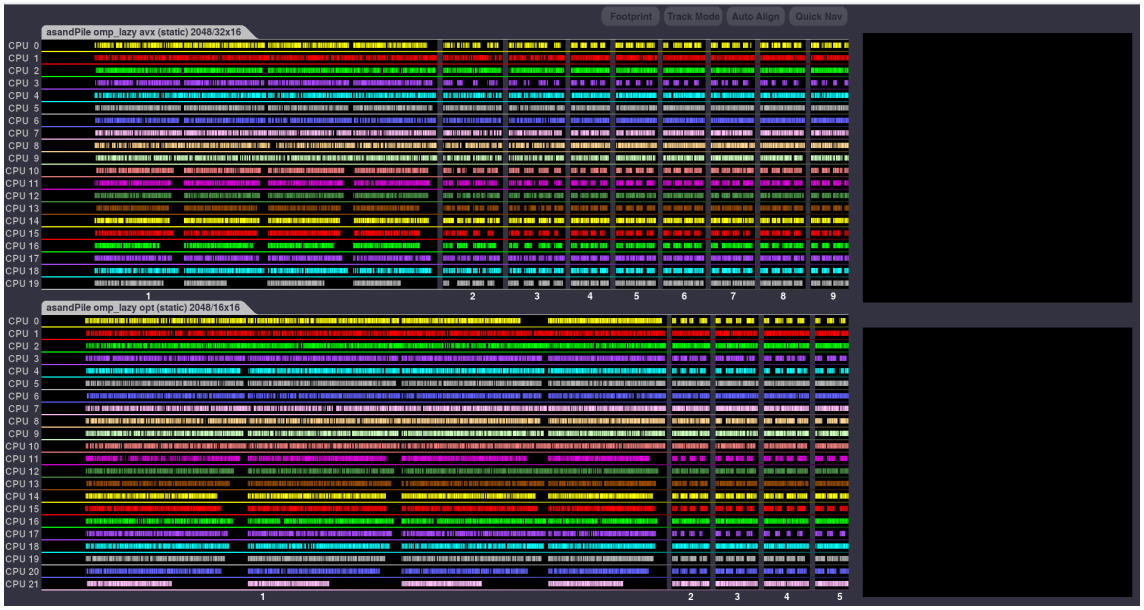

FIGURE 14 – Traces comparative du calcul `omp_lazy()` pour les premières itérations entre la version AVX et non-vectorisé (opt) avec leurs paramètres optimaux

FIGURE 15 – Traces comparative du calcul `omp_lazy()` pour une itération entre la version AVX et non-vectorisé (opt) avec leurs paramètres optimaux

3 Implémentation OpenCl

3.1 Présentation du code

Pour cette implémentation, dans un premier temps, nous nous sommes référés de l'implémentation du noyau **transpose** se trouvant dans le cours, nous avons employé différentes techniques, à savoir l'utilisant d'une tuile locale de `GPU_TILE_H x GPU_TILE_W` pour stocker le résultat du calcul de chaque tuile de la grille, l'ajout de 1 à GPU_TILE_W et faire le calcul directement dans le tableau(*out) sans passer par une tuile. Suite à cela, la plus optimisée a été celle faisant directement le calcul dans(*out) que nous allons présenter ci-dessous. De passage, nous notons que le traitement est réalisé que lorsqu'on est pas sur les bords.

```
static inline TYPE adjusted_self(TYPE id) {
    return id % 4;
}

static inline TYPE adjusted_neighbor(TYPE id) {
    return id / 4;
}

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

  if( x != 0 && x != DIM - 1 && y != DIM - 1 && y != 0 ) {
    out[y * DIM + x] = adjusted_self(in[y * DIM + x]) +
                       adjusted_neighbor(in[y * DIM + (x+1)]) +
                       adjusted_neighbor(in[y * DIM + (x-1)]) +
                       adjusted_neighbor(in[(y-1) * DIM + x]) +
                       adjusted_neighbor(in[(y+1) * DIM + x]);
  }
}
```

Listing 8 – Implémentation en OpenCl du noyau

3.2 Comparaison OpenCL et CPU

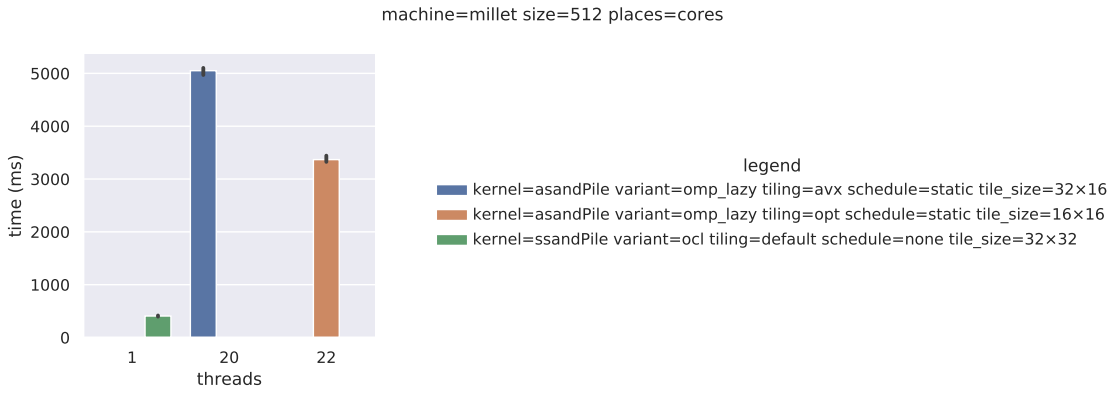

FIGURE 16 – Comparaison du temps moyen (n=3) entre l'exécution vectorisé, non-vectorisé et OpenCL avec les meilleurs paramètres d'éxécution

On se rend compte à notre très grande surprise que la version non-vectorisé (en orange) est plus rapide que la version utilisant AVX (en bleue). Nous pensons que cela est étroitement lié au problèmes de divergence qui se pose lors du calcul. Dans la première version on recalcule la valeur seulement ci la valeur de la case est supérieur ou égale à 4, tandis qu'en AVX c'est la loi des 16 mousquetaires qui régit si l'on va effectuer le calcul ou non. Cela n'apparaît pas lorsqu'on effectue un petit millier d'itération, mais à la fin, alors que plus de pixels se sont stabilisés, il y a un important sur-coût.

On constate aussi que la version sur carte graphique est bien plus rapide que celles sur les coeurs. Et encore, OpenCL est désavantagé. Puis ce qu'on ne compare pas les performances sur les mêmes noyaux.