Comment faire de jolis pâtés

Aguibou Barry Lucas Palacz

2 mars 2022

Première partie

Optimisations et parallélisme de base

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
7          table(out, i, j) = table(in, i, j) % 4;
8          table(out, i, j) += table(in, i + 1, j) / 4;
9          table(out, i, j) += table(in, i - 1, j) / 4;
10         table(out, i, j) += table(in, i, j + 1) / 4;
11         table(out, i, j) += table(in, i, j - 1) / 4;
12         if (table(out, i, j) >= 4)
13             diff = 1;
14         }
15
16     return diff;
17     }
```

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.

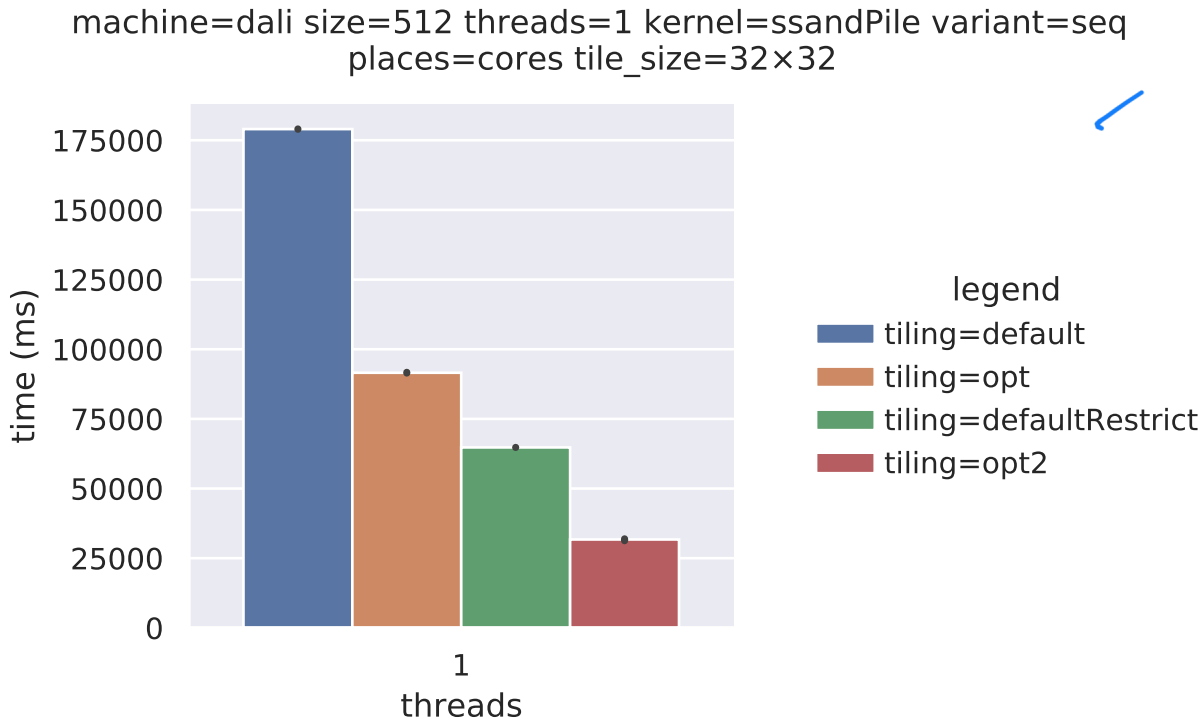

FIGURE 1 – Temps de calcul moyen (n=3) pour la version originale (default), la version originale mais TABLE est restrict (defaultRestrict), la première version optimisé (opt), la version optimisé finale, avec TABLE restrict et unroll-loops (opt2).

```
#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 de PA. Wacrenier

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.

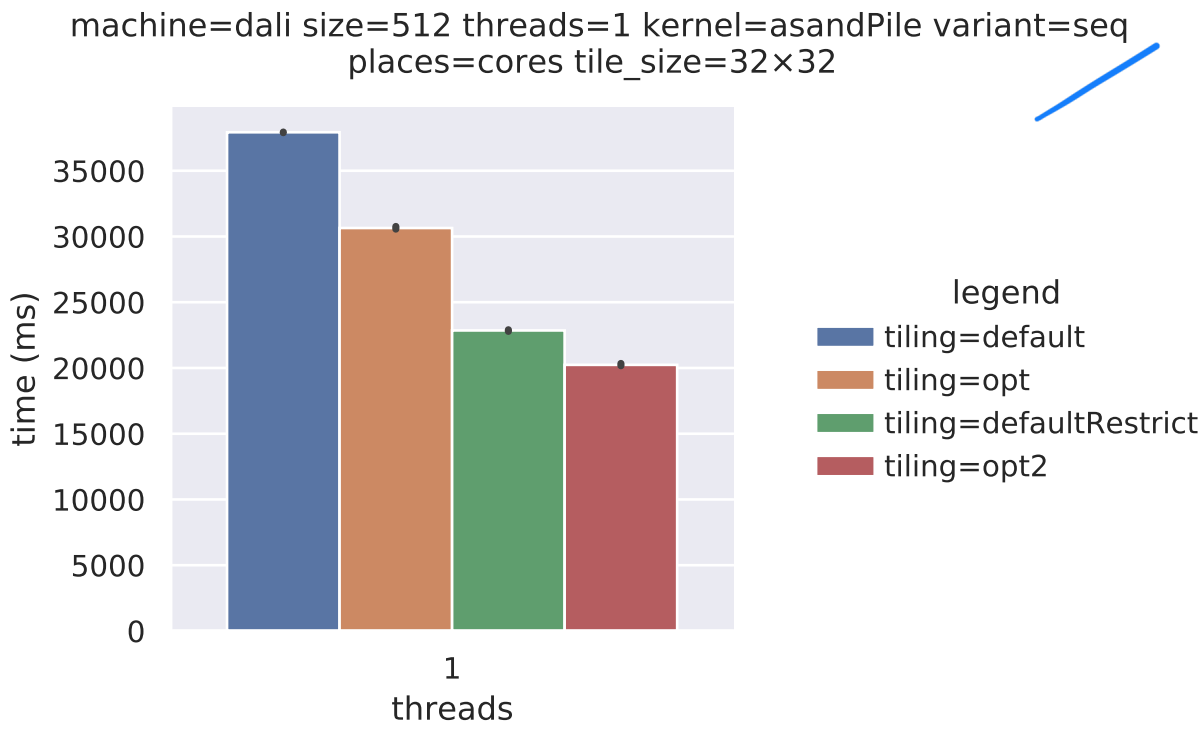

FIGURE 2 – Temps de calcul moyen (n=3) pour la version originale (default), la version originale mais TABLE est restrict (defaultRestrict), la première version optimisé (opt), la version optimisé finale, avec TABLE restrict et unroll-loops (opt2).

```
#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 Parallélisation de base

2.1 Avec la version synchrone du noyau

Suite aux différentes optimisations réalisées précedement et comme indiquer dans le sujet, nous avons implémenter trois versions de parallélisation pour ce noyau à savoir :

— **sandPile_compute_omp()** : dans cette version nous avons utiliser la directive #pragma omp parallel for avec une politique de distribution qui sera définit lors l'exécution du programme(runtime) tout en utilisant une seule boucle de parcours donc pas de tuile ;

```
unsigned ssandPile_compute_omp(unsigned nb_iter)
{
    for (unsigned it = 1; it <= nb_iter; it++)
    {
        int change = 0;
        int y;

#pragma omp parallel for schedule(runtime) shared(change, y)
        for (y = 0; y < DIM; ++y)
            change |=
                do_tile(1, y + (y == 0), DIM - 2,
                        1 - ((y + 1 == DIM) + (y == 0)),
                        omp_get_thread_num());
        swap_tables();
        if (change == 0)
          return it;
    }

    return 0;
}
```

Listing 5 – Parallélisation de base sans les tuiles

— **sandPile_compute_omp_tiled()** : cette parallélisation est similaire à la précédente à la différence que dans celle-ci le traitement est fait sur des tuiles et en plus on a rajouter la clause collapse() pour faire la fusion des deux boucles.

```
unsigned ssandPile_compute_omp_tiled(unsigned nb_iter) {
    for (unsigned it = 1; it <= nb_iter; it++) {
        int change = 0;
        int x,y;

#pragma omp parallel for collapse(2) shared(change, x, y)
    schedule(runtime)
        for (y = 0; y < DIM; y += TILE_H)
          for (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 == 0)
          return it;
    }
    return 0;
```

```
}
```

Listing 6 – Parallélisation de la variante tuilée

— **sandPile_compute_omp_taskloop()** : cette parallélisation est faite sur la base de la version tuilée avec l'utilisation de la clause taskloop qui consiste à créer de tâches qu'il y a autant de traitement.

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

    #pragma omp parallel
    #pragma omp single
    {
        for (unsigned it = 1; it <= nb_iter; it++)
        {
          int change = 0;

          #pragma omp taskloop collapse(2) shared(res, change)
          for (int x = 0; x < DIM; x += TILE_W)
            for (int y = 0; y < DIM; y += TILE_H)
              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()
     /* CPU id */);

          swap_tables();
          if (change == 0)
          {
            res = it;
            break;
          }
        }
    }
    return res;
}
```

Listing 7 – Parallélisation avec les tâches

2.2 Version synchrone du noyau

A Données

A.1 Optimisation du pipeline

```
machine;size;tilew;tileh;threads;kernel;variant;tiling;iterations
;schedule;places;label;arg;time
dali;512;32;32;1;ssandPile;seq;default;69190;;cores;;none;178952420
dali;512;32;32;1;asandPile;seq;default;34938;;cores;;none;37941265
dali;512;32;32;1;ssandPile;seq;default;69190;;cores;;none;179024647
dali;512;32;32;1;asandPile;seq;default;34938;;cores;;none;37942565
dali;512;32;32;1;ssandPile;seq;default;69190;;cores;;none;179053874
dali;512;32;32;1;asandPile;seq;default;34938;;cores;;none;37895301
dali;512;32;32;1;ssandPile;seq;defaultRestrict;69190;;cores;;none;64757375
dali;512;32;32;1;ssandPile;seq;opt2;69190;;cores;;none;31946655
dali;512;32;32;1;asandPile;seq;defaultRestrict;34938;;cores;;none;22874312
```

```
dali;512;32;32;1;asandPile;seq;opt2;34938;;cores;;none;20172211
dali;512;32;32;1;ssandPile;seq;defaultRestrict;69190;;cores;;none;64768967
dali;512;32;32;1;ssandPile;seq;opt2;69190;;cores;;none;31938320
dali;512;32;32;1;asandPile;seq;defaultRestrict;34938;;cores;;none;22905474
dali;512;32;32;1;asandPile;seq;opt2;34938;;cores;;none;20343324
dali;512;32;32;1;ssandPile;seq;defaultRestrict;69190;;cores;;none;64761430
dali;512;32;32;1;ssandPile;seq;opt2;69190;;cores;;none;31282851
dali;512;32;32;1;asandPile;seq;defaultRestrict;34938;;cores;;none;22784928
dali;512;32;32;1;asandPile;seq;opt2;34938;;cores;;none;20198548
dali;512;32;32;1;ssandPile;seq;opt;69190;;cores;;none;91713571
dali;512;32;32;1;asandPile;seq;opt;34938;;cores;;none;30562081
dali;512;32;32;1;ssandPile;seq;opt;69190;;cores;;none;91458392
dali;512;32;32;1;asandPile;seq;opt;34938;;cores;;none;30762735
dali;512;32;32;1;ssandPile;seq;opt;69190;;cores;;none;91509505
dali;512;32;32;1;asandPile;seq;opt;34938;;cores;;none;30609857
```