Programmation Parallèle - Tas de sable abéliens

Etape 3 (4.5 et 4.7.1)

Tarrieu Odrian, Corentin Drezen, Mouhamadou Mansour Gueye

Mars/Avril 2024

1 Avant propos

On a réorganisé ***sandPile.c*** en plusieurs fichiers pour facilité notre lisibilité. On a fait ça avec plusieurs ***.h*** bien que l'on soit conscient que ce n'est pas moral de mettre des définitions dans les en-têtes.

On a donc dans le dossier *kernel/c/*:

- **sandPile.c** : seulement *ILP Optimization*
- **sandPile_omp_ssand.h** : *OpenMP synchronous version*
- **sandPile_omp_asand.h** : *OpenMP asynchronous version*
- **sandPile_omp_ssand.h** : *Lazy synchronous version*
- **sandPile_omp_asand.h** : *Lazy asynchronous version*
- **sandPile_avx.h** : *Implementations AVX*

2 Fix lazy implementation

On a réussi a corriger notre version paresseuse synchrone qui ne fonctionnait pas au rendu précédent.

On a modifié une ligne de notre ***do_tile*** optimisé pour que la version synchrone puisse prendre en compte le cas où la modification d'une tuile n'était pas détecté en vérifiant uniquement si une chute de sable allait avoir lieu à la prochaine itération:

```
int ssandPile_do_tile_lazy(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++)
    {
      TYPE* cell_in = table_cell(TABLE, in, i, j);
      TYPE* cell_out = table_cell(TABLE, out, i, j);

      *cell_out = table(in, i, j) % 4
                + (table(in, i, j - 1) / 4)
                + (table(in, i, j + 1) / 4)
                + (*(cell_in - DIM) / 4)  // table(in, i - 1, j) / 4
                + (*(cell_in + DIM) / 4); // table(in, i + 1, j) / 4

      if ( *cell_in != *cell_out )//(*cell_out >= 4) modifié pour tuile stagnate
      {
        diff = 1;
      }
    }

  return diff;
}
```

Pour que la détection des tuiles stagnantes on avait simplement besoin d'un tableau booléen de taille double plutôt que de faire un premier passage et comparer la tuile courante.

De plus, il n'y avait pas nécessairement besoin d'essayer de vérifier la stagnation des tuiles voisines avant de marquer une tuile comme stagnante si à la réactivation d'une tuile on marque toutes ses voisines comme actives.

```
unsigned ssandPile_compute_lazy(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++)
  {
    int change = 0;
  #pragma omp parallel for collapse(2) schedule(runtime) reduction(|:change)
    for (int y = 0; y < DIM; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
      {
        if(is_steady(in, y, x) && is_steady(out, y, x) ) continue;

        int diff = do_tile(x + (x == 0), y + (y == 0),
                    TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                    TILE_H - ((y + TILE_H == DIM) + (y == 0)));

        change |= diff;

        if(diff == 0)// && (firstCheck || has_active_neighbors(in, y, x)))//
        {
          #pragma omp atomic
          is_steady(in, y, x) |= true;
        }
        else
        {
          if (!(x == 0))
            is_steady(out, y, x - TILE_W) &= false;

          if (!(x + TILE_W == DIM))
            is_steady(out, y, x + TILE_W) &= false;

          if (!(y + TILE_H == DIM))
            is_steady(out, y + TILE_H, x) &= false;

          if (!(y == 0))
            is_steady(out, y - TILE_H, x) &= false;
        }
      }

    swap_tables();
    if (change == 0) return it;
  }

  return 0;
```

Cependant une réactivation plus précise serait de réactiver uniquement les tuiles avec en bordure une tuile voisine avec du sable qui va s'écrouler.

On a donc aussi essayé de de réaliser une version comme cela (*lazy1*) en se s'appuyant sur les informations de *do_tile* mais elle s'avère plus coûteuse comme on fait des comparaison pour chaque pixels.

On observe néanmoins une nette amélioration des performance avec l'option *spirals* et il serait peut être possible de réduire le coût de détection en s'aidant d'un tableau enregistrant plus d'informations pour chaque tuiles.

Plus tard, on a découvert un bug de calcul de d'addresse de notre **lazy** pour les tuiles rectangulaire de hauteur $1 < x < DIM/4$ et de largeur DIM. Il n'est pas en lien avec notre implementation AVX car il peut se produire sans.

3 Fix bug async parallelisation

Le bug de notre version asynchrone mentionné dans le rapport précédant venait de la non prise en charge de tuiles rectangulaire de longueur égale à l'image.

Comme on découpait notre image en tuiles d'un damier cela posait problème.

```
for (int y = 0; y < DIM; y += 2*TILE_H)
      for (int x = 0; x < DIM; x += 2*TILE_W){
        change |=
            do_tile(x + (x == 0), y + (y == 0),
                    TILE_W - (x == 0),
                    TILE_H - (y == 0)  - (y == 0 && TILE_H == DIM) );//TILE_H - (y == 0));

        if(TILE_H == DIM) continue; //pas bon iter 1:

        change |=
             do_tile(x + TILE_W, y + TILE_H,
                     TILE_W - (x + 2*TILE_W == DIM),
                     TILE_H - (y + 2*TILE_H == DIM));
      }
```

4 AVX implementation - synchrone

On a d'abord essayer de faire une 'traduction' fidèle à notre *do_tile* synchrone en utilisant des vecteurs de 256 bits.

```
int ssandPile_do_tile_avx_256(int x, int y, int width, int height)
{
  int diff = 0;
  const __m256i THREE_VEC = _mm256_set1_epi32(3);

  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j += AVX_VEC_SIZE_FLOAT) {

      __m256i cell_in, cell_out, cell_left, cell_right, cell_top, cell_bottom;

      cell_in = _mm256_loadu_si256 ((__m256i *)&table(in, i, j));
      cell_out = _mm256_loadu_si256 ((__m256i *)&table(out, i, j));
      cell_left = _mm256_loadu_si256 ((__m256i *)&table(in, i, j - 1));
      cell_right = _mm256_loadu_si256 ((__m256i *)&table(in, i, j + 1));
      cell_top = _mm256_loadu_si256 ((__m256i *)&table(in, i + 1, j));
      cell_bottom = _mm256_loadu_si256 ((__m256i *)&table(in, i - 1, j));

      //cell_out = cell_in % 4 (= &3):
      cell_out = _mm256_and_si256(cell_in, THREE_VEC);

      // neighbors /4 (= >>2):
      cell_left = _mm256_srli_epi32(cell_left, 2);
      cell_right = _mm256_srli_epi32(cell_right, 2);
      cell_top = _mm256_srli_epi32(cell_top, 2);
      cell_bottom = _mm256_srli_epi32(cell_bottom, 2);

      // cell_out += neighbors / 4 :
      cell_out = _mm256_add_epi32(cell_out, cell_left);
      cell_out = _mm256_add_epi32(cell_out, cell_right);
      cell_out = _mm256_add_epi32(cell_out, cell_top);
      cell_out = _mm256_add_epi32(cell_out, cell_bottom);

      _mm256_storeu_si256((__m256i *)&table(out, i, j), cell_out);

      ///diff = cell_in != cell_out :
      __mmask8 mask = _mm256_cmpneq_epu32_mask(cell_in, cell_out);// cell_in != cell_out : 0 modif -> 00000000 (a
      diff = (int)mask;//__mmask8 = uint_8
    }

  return  diff;
}
```

On l'a ensuite adapté à des vecteurs de 512 bits. On a à première vue pas observé de réel différence de performance mais voici une expérience met en lumière l'avantage d'utiliser des plus grand vecteurs:

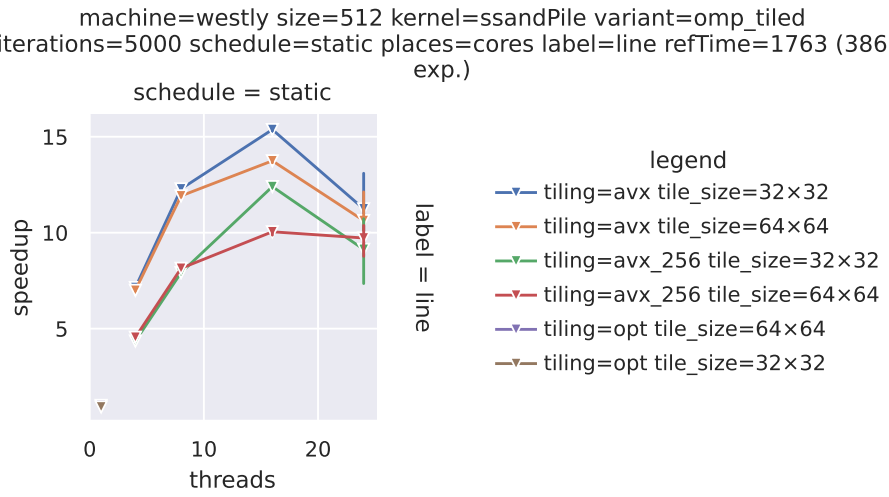

Figure 1: avx512(appelée 'avx') vs avx256

La bonne taille des vecteurs par rapport au cache (512bits = une ligne dans le cache) explique la meilleur efficacité de la version avec des mm512.

Pour nous faciliter la tache on a d'abord fait géré le traitement des bords par un *compute* dédié et il nous a semblé que seule le bord droit nécessitait d'être traité différemment comme le problème semblait apparaître à l'itération 14 sur le bords droit et se propagé ensuite au reste de l'image.

On suppose que problème venait du fait que lors du calcul des derniers pixels du dernier vecteur on lisait les pixels du bords que l'on ne lit pas d'habitude.

Pour régler cela en instruction avx on a conditionné l'addition qui dépend du pixel droit lorsqu'il s'agit d'une tuile en bordure droite à l'aide d'un masque et de l'instruction ***_mask_add_epi32*** qui pour chaque epi32 du vecteur additionne deux arguments si mask[i] est vrai et l'assigne à la valeur d'un autre argument sinon.

On a trouvé que pour des vecteurs de 256 bits, le masque ci-dessus permettait d'ignorer les deux dernier éléments du vecteur pour l'addition. Cependant on aurait pu penser que le sens de la représentation des éléments du vecteur était inversé et qu'il fallait ignorer seulement le dernier epi32.

```
int ssandPile_do_tile_avx(int x, int y, int width, int height)
{
    int diff = 0;
    const __m512i THREE_VEC = _mm512_set1_epi32(3);

    for (int i = y; i < y + height; i++)
    {
        for (in j = x; j < x + width; j += AVX512_VEC_SIZE_FLOAT) // Utilisation de pas de 16 pour AVX-512
        {
            __m512i cell_in = _mm512_loadu_si512((__m512i *)&table(in, i, j));
            __m512i cell_out = _mm512_loadu_si512((__m512i *)&table(out, i, j));
            __m512i cell_left = _mm512_loadu_si512((__m512i *)&table(in, i, j - 1));
            __m512i cell_right = _mm512_loadu_si512((__m512i *)&table(in, i, j + 1));
            __m512i cell_top = _mm512_loadu_si512((__m512i *)&table(in, i + 1, j));
            __m512i cell_bottom = _mm512_loadu_si512((__m512i *)&table(in, i - 1, j));

            // Calcul de cell_out = cell_in % 4 (= &3)
            cell_out = _mm512_and_epi32(cell_in, THREE_VEC);

            // + neighbors / 4
            cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_left, 2));
            cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_top, 2));
            cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_bottom, 2));

            if(j != DIM - AVX512_VEC_SIZE_FLOAT)
            {
              cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_right, 2));
            }
            else
            {
              cell_out = _mm512_mask_add_epi32( cell_out, 0x3FFF, cell_out, _mm512_srli_epi32(cell_right, 2));
            }


            // Stockage du résultat
            _mm512_storeu_si512((__m512i *)&table(out, i, j), cell_out);

            // Comparaison de cell_in et cell_out pour détecter les modifications
            __mmask16 mask = _mm512_cmpneq_epu32_mask(cell_in, cell_out);
            diff = mask; //0 si cell_in != cell_out

        }
    }

    return diff;
}
```

En combinant ce masque à un masque qui vérifie que l'on se trouve sur la dernière tuile, on peut gérer le bord droit sans *_compute* dédié et aussi sans *if* qui pourrait gêner le compilateur.

On finalement utilisé un ***_blend_*** pour réaliser cela (on a penser à utiliser ***_kand***):

```
int ssandPile_do_tile_avx(int x, int y, int width, int height)
{
    unsigned diff = 0;
    const __m512i THREE_VEC = _mm512_set1_epi32(3);
    const __m512i LAST_VEC_J = _mm512_set1_epi32(DIM - AVX512_VEC_SIZE_FLOAT);

    for (int i = y; i < y + height; i++)
    {
        for (int j = x; j < x + width; j += AVX512_VEC_SIZE_FLOAT) // Utilisation de pas de 16 pour AVX-512
        {
            __m512i cell_in = _mm512_loadu_si512((__m512i *)&table(in, i, j));
            __m512i cell_out = _mm512_loadu_si512((__m512i *)&table(out, i, j));

            __m512i cell_left = _mm512_loadu_si512((__m512i *)&table(in, i, j - 1));
            __m512i cell_right = _mm512_loadu_si512((__m512i *)&table(in, i, j + 1));
            __m512i cell_top = _mm512_loadu_si512((__m512i *)&table(in, i + 1, j));
            __m512i cell_bottom = _mm512_loadu_si512((__m512i *)&table(in, i - 1, j));

            // Calcul de cell_out = cell_in % 4 (= &3) + neighbors / 4
            cell_out = _mm512_and_epi32(cell_in, THREE_VEC);
            cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_top, 2));
            cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_bottom, 2));
            cell_out = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_left, 2));

            const __m512i J_VEC = _mm512_set1_epi32(j);

            // = if(j != DIM - AVX512_VEC_SIZE_FLOAT) :
            __mmask16 mask_last = _mm512_cmpeq_epu32_mask(J_VEC, LAST_VEC_J);

            __m512i regular = _mm512_add_epi32(cell_out, _mm512_srli_epi32(cell_right, 2));//if

            __m512i last_tile_case = _mm512_mask_add_epi32(cell_out,                    //else
                                                           0x3FFF,
                                                           cell_out,
                                                           _mm512_srli_epi32(cell_right, 2));

            cell_out = _mm512_mask_blend_epi32(mask_last, regular, last_tile_case);

            // Stockage du résultat
            _mm512_storeu_si512((__m512i *)&table(out, i, j), cell_out);

            // Comparaison de cell_in et cell_out pour détecter les modifications
            __mmask16 mask = _mm512_cmpneq_epu32_mask(cell_in, cell_out);
            diff |= mask;
        }
    }

    return diff;
}
```

Il semble que seulement les tuiles carrés multiples de 512 ou 256 ou les tuiles tel que $TILE\_H * TILE\_W = nb\_bits(VEC)$ fonctionnent avec nos version SIMD. On suppose que cela est normal, qu'autrement les pixels ne sont pas lus aux bonnes addresses comme on en traite $\_VEC\_SIZE\_$ chacun à chaque tour de boucle.

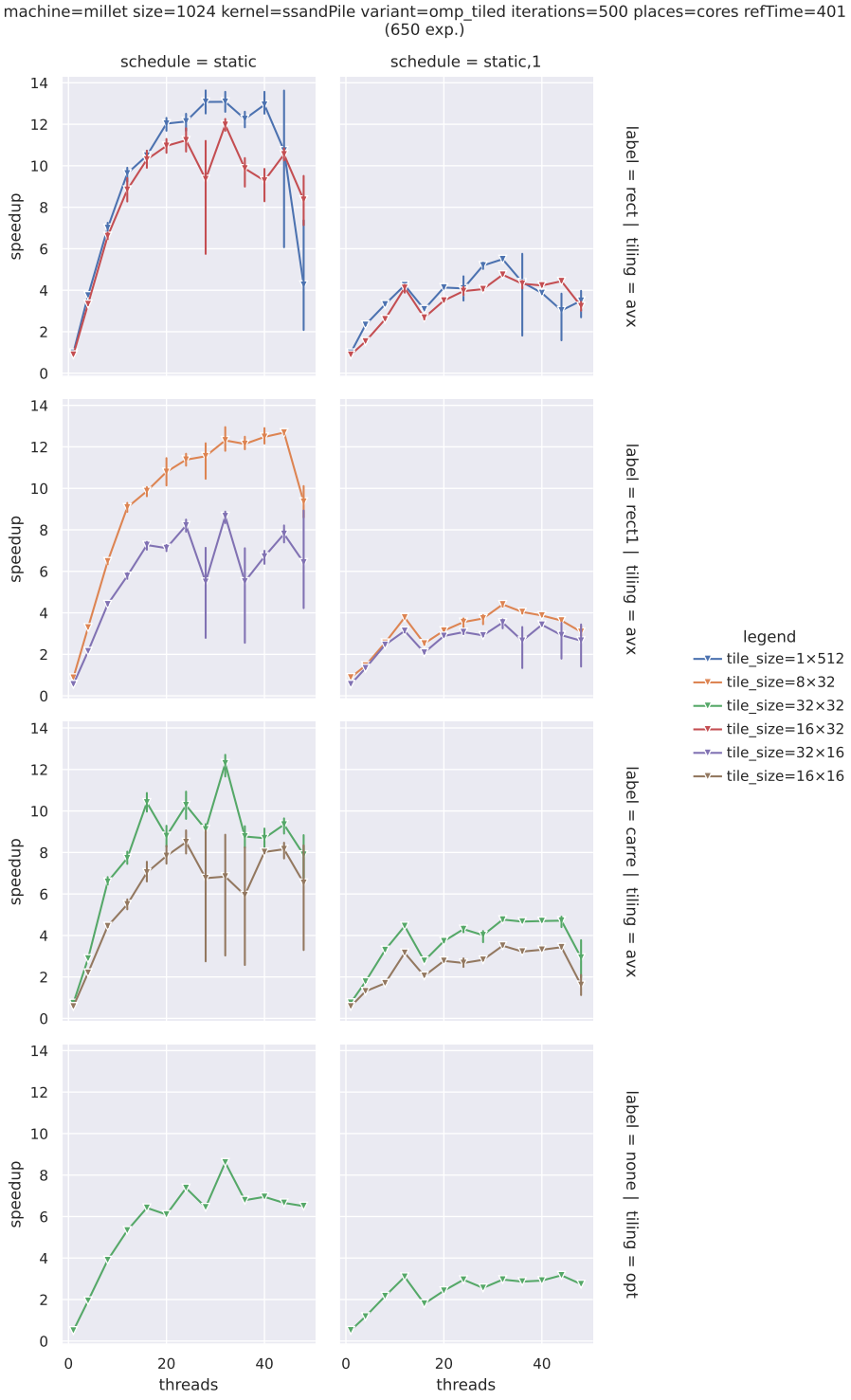

Figure 2: Comparaisons des tuiles compatibles à défaut d'un heatmap

Avec l'expérience ci dessus on remarque que la meilleur taille de tuile pour la version avec des vecteur 512bits est une tuile en ligne 1x512. Cela semblerait cohérent avec le fait qu'un vecteur 512bits (16*32) correspond à une ligne dans dans le cache cependant il s'agit de 512*4*32 bits pour une tuile et non pas 512bits.

Néanmoins la linéarité de la tuile en mémoire et la divisibilité de sa taille par 512 doit contribuer en plus de la taille de l'image traité (1024).

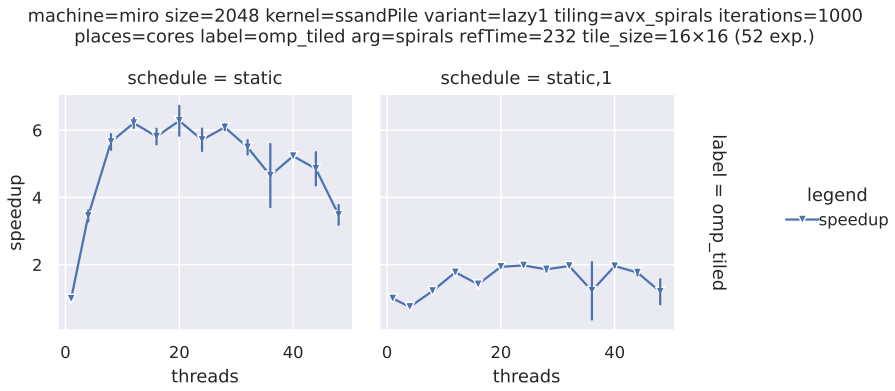

Figure 3: Speedup ssandPile Spiral AVX 2048 16x16

Pour l'évaluation 2048 sur le calcule du modèle spiral avec la version synchrone SIMD, on a réutilisé notre ***lazy1*** qui était plus performant avec spirals mais qui nécessitait un do_tile différent qui réactives les tuile (ce qui n'est pas sensé être aussi performant que réactiver dans le compute en temps normal). On a donc retranscrit un do_tile_ avx correspondant. Cepandant on a a pas encore supprimé les ***ifs*** qui font des divergence qui freinent sans doute la vectorisation.

Nous avons remarqué que toutes les tailles de tuile autres de 16x16 ne fonctionnent pas. Peut être parce que nos vecteurs contiennent 16 valeurs. De plus la taille d'un vecteur 512bits (16*32) correspond à une ligne dans dans le cache.

Nous avons donc réalisé un speedup avec le tuilage 16x16, et nous avons remarqué un pic à 20 threads.

En testant avec 17, 18 et 19 thread, nous avons déduis que le meilleur nombre de thread est 18.

On observe que les traces possèdent une taille homogène ce qui montre l'activité homogène de tout les threads l'éxécution du test. (figure 4)

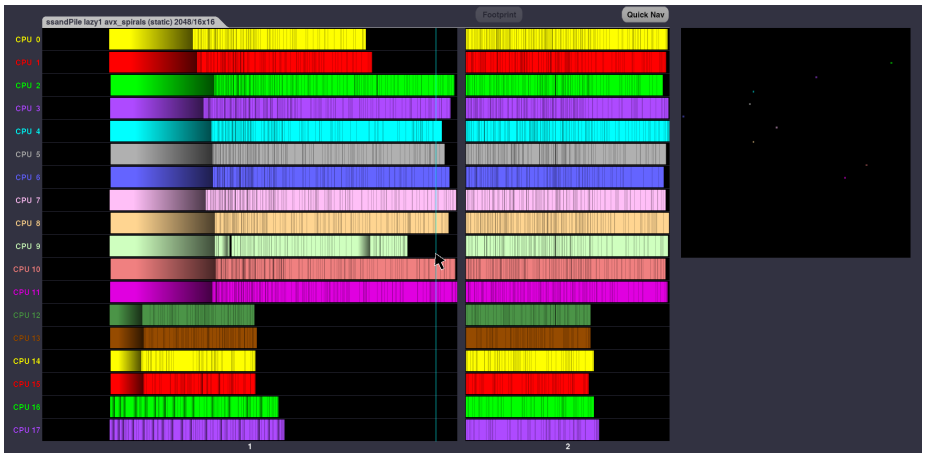

Figure 4: Traces ssandPile Spiral AVX 2048 16x16

5 AVX implementation - asynchrone

Pour la version asynchrone on a essayé de faire comme dans l'énoncé mais on a pas réussi à résoudre le problème du bord droit sans avoir recours à la version non SIMD.

Il y a un problème de synchronization qui ne vient pas de notre implémentation SIMD mais de notre parallélisation de la fonction asand_compute_omp. Car c'est probable qu'en utilisant un damier cela pose problème pour la vectorisation avec les pixels traités qui se retrouvent non-alignés par rapport aux différents vecteurs.

Ainsi pour le sandCheck 512 il peut arriver, rarement qu'il y ai un bug de synchronisation qui est moins courant avec un nombre de threads ≤ 4

On a essayé d'utiliser des directives OpenMP *safelen* qui empecherait les threads d'entrer en concurrence sur une certaine distance, et aussi *aligned* mais elle dépendent d'une directive qui ne semble pas être approprié avec ce que l'on a fait.

On a pas factorisé les *_load_* et *_stores_*, on suppose que l'on pourrait accumuler les atable[i][j - 1] et atable[i][j + 1].

```
int asandPile_do_tile_avx(int x, int y, int width, int height)
{
  if((x == AVX512_VEC_SIZE_FLOAT) || (x == DIM - AVX512_VEC_SIZE_FLOAT)
  || (y == AVX512_VEC_SIZE_FLOAT) || (y == DIM - AVX512_VEC_SIZE_FLOAT)) {
    return asandPile_do_tile_opt1(x, y, width, height);
  }

  unsigned diff = 0;
  const __m512i THREE_VEC = _mm512_set1_epi32(3);
  const __m512i ZERO_VEC = _mm512_set1_epi32(0);

  //TYPE* ptr = atable_cell(TABLE, y - 1, x);
  //#pragma omp simd collapse(2) reduction(|:diff) safelen(AVX512_VEC_SIZE_FLOAT*5) aligned(ptr:32*5)// private(Dk
  for (int i = y; i < y + height; i++){
    for (int j = x; j < x + width; j += AVX512_VEC_SIZE_FLOAT)
    {

      __m512i cells = _mm512_loadu_si512((__m512i *)&atable(i, j));
      __m512i last_cells = cells;

      __m512i cells_top = _mm512_loadu_si512 ((__m256i *)&atable(i + 1, j));
      __m512i cells_bottom = _mm512_loadu_si512((__m256i *)&atable(i - 1, j));

      __m512i D = _mm512_srli_epi32(cells, 2);

      // %4 + D<<1 +D>>1 :
      cells = _mm512_and_epi32(cells, THREE_VEC);
      // >> 1 (plus rapide avec mask ici)
      cells = _mm512_add_epi32(cells, _mm512_mask_alignr_epi32(ZERO_VEC, 0x7FFF, ZERO_VEC, D, 1));
      // << 1
      cells = _mm512_add_epi32(cells, _mm512_alignr_epi32(D, ZERO_VEC, AVX512_VEC_SIZE_FLOAT - 1));

      cells_top = _mm512_add_epi32(cells_top, D);
      cells_bottom = _mm512_add_epi32(cells_bottom, D);

      TYPE Dk[16] = { 0 };
      _mm512_storeu_epi32(&Dk, D);

      //Tj-1,i += D[0]:
      atable(i, j - 1) += Dk[0];

      //Tj+1+k,i += D[k] :
      atable(i, j + AVX512_VEC_SIZE_FLOAT) += Dk[AVX512_VEC_SIZE_FLOAT - 1];

      _mm512_storeu_si512((__m512i *)&atable(i - 1, j), cells_bottom);
      _mm512_storeu_si512((__m512i *)&atable(i, j), cells);
      _mm512_storeu_si512((__m512i *)&atable(i + 1, j), cells_top);

      __mmask16 mask = _mm512_cmpneq_epu32_mask(cells, last_cells);
      diff |= mask;
    }
  }

  return diff;
}
```

Pour le sandCheck ”8192 -a alea -k asandPile” on a décider d’utiliser ssandPile car on sait que cette operation cera plus rapide avec une implementation paresseuse mais

notre lazy asynchrone ne fonctionne pas en plus d'être basé sur une parralelisation incompatible avec la vectorization SIMD.

6 OpenCL Implementation

Voici le code de notre implémentation. Nous obtenons environs un temps de 368.5ms. L'implémentation OpenCL est significativement plus rapide que l'implémentation AVX.

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

    int pos = y * DIM + x;

    if (x != 0 && y != 0 && x != DIM - 1 && y != DIM - 1) {
        unsigned cell_out = in[pos] % 4
                          + in[pos + 1] / 4
                          + in[pos - 1] / 4
                          + in[pos + DIM] / 4
                          + in[pos - DIM] / 4;
        out[pos] = cell_out;
    }
}
```

7 Traces

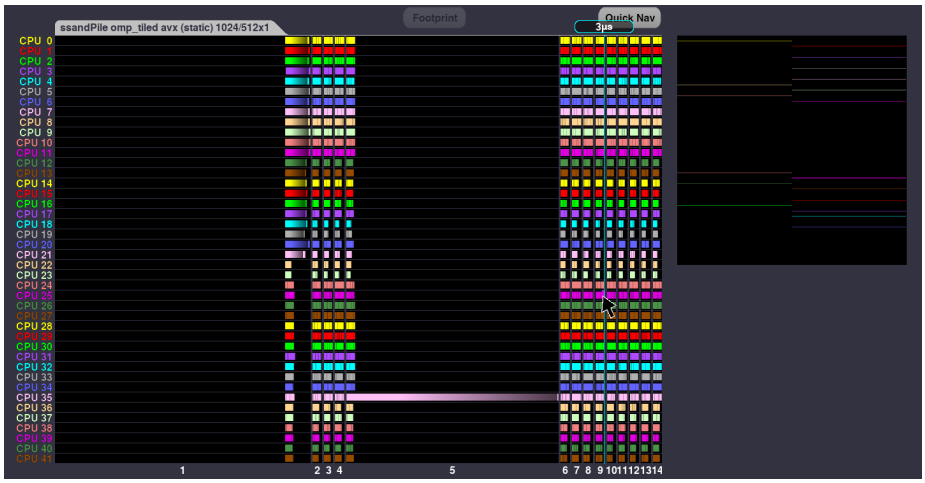

Figure 5: ssanPile omp_tiled avx (static) 1024/512x1

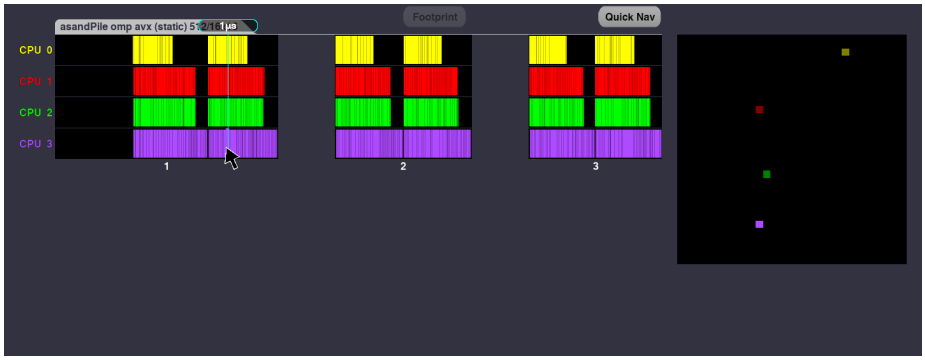

Figure 6: assandPile omp avx (static) 512/16x16

Figure 7: ssandPile lazy avx (static) 8192/16x16

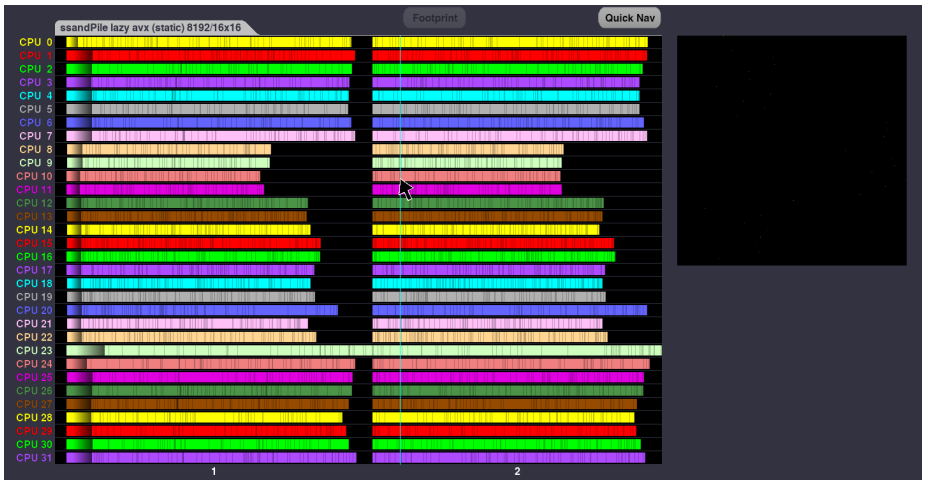

Figure 8: ssandPile lazy avx (static) 8192/16x16