Version avancée (GPU + OpenMP)

Projet : Tas de sable abéliens

ind-46

Aguibou Barry

1 mai 2022

Programmation Parallèle

1 OpenCL + OpenMP Implementation

1.1 Présentation du code

1.1.1 Détection de la terminaison

Le premier point abordé dans ce rendu est la terminaison du programme, pour cela j'ai utilisé un buffer (`ssand_buffer`) que j'alloue lors de l'initialisation de ma fonction avec une taille **(DIM * DIM)** que j'ai trouvé plus optimale que **(DIM / GPU_TILE_W) * (DIM / GPU_TILE_H)** car cette dernière nécessite un certain nombre de calcul pour retrouver la tuile correspondante à chaque cellule de la grille. En se référant du code du noyau *"scrollup"* j'ai su comment se servir de ce buffer côté GPU que je transfère par la suite dans un tableau `(TABLE_BUFF)` initialiser également lors de l'initialisation de la fonction avec la même dimension que le buffer. Ce transfert est réalisé au moyen de `"clEnqueueReadBuffer(...)"`. Après ce transfert, il suffit de vérifier le contenu des cases de ce tableau pour savoir si on est arrivé à la stabilisation. Le transfert étant très coûteux, j'ai employé un mécanisme qui consiste à faire ce transfert que lorsqu'on atteint un certain nombre d'itérations qui est calculé en fonction de la dimension de l'image. Après plusieurs tentatives pour trouver une formule qui donnera exactement le nombre d'itérations au bout duquel on arrive à la stabilisation sans succès, j'ai opté finalement de trouver une valeur approchée qui sera applicable sur n'importe quelle dimension. Pour une image de taille 32, ce nombre est égal à `DIM/32 * 7 * DIM` et pour toutes autres dimensions, ce nombre est `DIM/32 * 8 * DIM`. Une fois ce nombre atteint, la lecture du buffer n'est pas faite de suite, il y a encore un autre teste qui consiste à contrôler le transfert de sorte que cela soit fait à chaque fois que le nombre d'itérations atteint 30% de la dimension de l'image, cela permet de faire le moins de transfert possible et permet de se rapprocher au temps d'exécution en utilisant l'itération exacte de la stabilisation de l'image. voir(Listing 3 et Listing 2)

```
1  #define VAL1 7 // Use for image size 32
2  #define VAL2 8  // Use for any other image size
3  #define DEFAULT_DIM 32
4  #define MAX_ITERATIONS (DIM == DEFAULT_DIM ? (DIM/DEFAULT_DIM * VAL1 * DIM) : (DIM/
       DEFAULT_DIM * VAL2 * DIM))
5  #define THIRTY_PERCENT (DIM * 30) / 100 // Use to compare after "MAX_ITERATIONS" is
       reached for copying "ssan_buffer" into "TABLE"
6
7
8  unsigned ssandPile_invoke_ocl_ssand (unsigned nb_iter) {
9    ...
10
11   int diff = 0;
```

```
12   for (unsigned it = 1; it <= nb_iter; it++) {
13     ++itVal;
14
15     // Set kernel arguments
16     err = 0;
17     ...
18     err |= clSetKernelArg (compute_kernel, 2, sizeof (cl_mem), &ssand_buffer);
19     err |= clSetKernelArg (compute_kernel, 3, sizeof (unsigned int), &itVal);
20     check (err, "Failed to set kernel arguments");
21
22     // Launch GPU kernel
23     ...
24
25     // Swap buffers
26     ...
27
28     if(itVal > MAX_ITERATIONS) {
29       if(++percent == THIRTY_PERCENT) {
30         int size = DIM * DIM;
31         err = clEnqueueReadBuffer (queue, ssand_buffer, CL_TRUE, 0,
32                         sizeof (unsigned) * size, TABLE_BUFF, 0, NULL, NULL);
33         check (err, "Failed to read to buffer");
34
35         percent = 0;  int change = 0;
36         for(int i = 0; i < size; i++)
37           change |= TABLE_BUFF[i];
38         if(!change) {
39           diff = it;
40           break;
41         }
42       }
43     }
44   }
45   ...
46
47   return diff;
48 }
```

Listing 1 – Implémentation openCL avec la detection de la terminaison "sandPile.c"

```
////////////////////////// opencl version only with program shutdown(ocl_ssand)
__kernel void ssandPile_ocl_ssand (__global unsigned *in, __global unsigned *out,
    __global unsigned *buffer, unsigned offset, unsigned iter) {

  int x = get_global_id (0);
  int y = get_global_id (1);

  int valX, valY;

  valX = x==0 ? (x==0 ? x+1 : x) : (x==DIM-1 ? (x==DIM-1 ? DIM-2 : x+1) : x);
  valY = y==0 ? (y==0 ? y+1 : y) : (y==DIM-1 ? (y==DIM-1 ? DIM-2 : y+1) : y);

  out[valY * DIM + valX] = adjusted_self(in[valY * DIM + valX]) +
                           adjusted_neighbor(in[valY * DIM + (valX+1)]) +
                           adjusted_neighbor(in[valY * DIM + (valX-1)]) +
                           adjusted_neighbor(in[(valY-1) * DIM + valX]) +
                           adjusted_neighbor(in[(valY+1) * DIM + valX]);

  if(iter > MAX_ITERATIONS)
    buffer[valY*DIM+valX] = (in[valY*DIM+valX] != out[valY*DIM+valX]);
}
```

Listing 2 – Implémentation openCL avec la detection de la terminaison "ssandPile.cl"

1.1.2 Implémentions OpenCL + OpenMP

Ce second point consiste à implémenter une version permettant de repartir la charge de travail entre le GPU et le CPU. Inspiré de l'implémentions hybrid du noyau mandel qui nous est fourni, j'ai utilisé la même technique à la différence que dans celui-ci le CPU et GPU doit se partager respectivement la dernière et la première ligne de traitement pour la synchronisation du travail, c'est-à-dire que lorsque le CPU procède au traitement de sa dernière ligne, sachant qu'à chaque éboulement les modifications portent sur trois lignes alors il aura besoin des informations de la première ligne traitée par le GPU pour faire son traitement et le GPU en faisant le traitement de sa première ligne il aura besoin de la dernière ligne traitée par le CPU donc ici à chaque fin d'exécution des deux parties j'utilise `clEnqueueReadBuffer(...)` pour lire la première ligne que le GPU vient de traiter el`EnqueueWriteBuffer(...)` pour écrire la dernière ligne que le CPU vient de traiter. J'ai implémenté deux versions, la première avec une répartition 50/50 entre le CPU et GPU qui marche correctement (renvoie la bonne image en sortie) sur n'importe quelle dimension et la deuxième avec une répartition dynamique qui marche aussi, mais qui est un peu moins performante que la première, car en plus de se partager les deux lignes de séparation, à chaque balancement il faudra également lire ou écrire la tuile de séparation avant le balancement cette lecture et écriture étant coûteux. Par manque de temps je n'ai pas pu tester si cette deuxième version marche correctement pour n'importe quelle dimension, mais ce qui est sûr c'est qu'elle marche pour une dimension(512 et 1024).

Ci-dessous le code la version avec une répartition 50/50 entre le CPU et le GPU.

```
1  static unsigned cpu_y_part; // The part that the CPU processes
2  static unsigned gpu_y_part; // The part that the GPU processes
3
4  void ssandPile_init_ocl_ssand_hybrid (void) {
5    ssandPile_init ();
6    if (GPU_TILE_H != TILE_H)
7      exit_with_error ("CPU and GPU Tiles should have the same height (%d != %d)",
8                       GPU_TILE_H, TILE_H);
9    // Start with fifty-fifty
10   cpu_y_part = (NB_TILES_Y / 2) * GPU_TILE_H;
11   gpu_y_part = DIM - cpu_y_part;
12   int size = DIM * gpu_y_part;
13   ssand_buffer = clCreateBuffer (context, CL_MEM_READ_WRITE,
14                                  sizeof (TYPE) *size, NULL, NULL);
15   if (!ssand_buffer)
16     exit_with_error ("Failed to allocate buffer for exit");
17   TABLE_BUFF = (TYPE *) malloc(size * sizeof(TYPE));
18   if (!TABLE_BUFF)
19     exit_with_error ("Failed to allocate table for exit");
20 }
21
22 // Only called when --dump or --thumbnails is used
23 void ssandPile_refresh_img_ocl_ssand_hybrid ()
24 {
25   cl_int err;
26   err = clEnqueueReadBuffer (queue, cur_buffer, CL_TRUE, cpu_y_part*DIM*sizeof(
      unsigned), sizeof (unsigned) * DIM * (gpu_y_part-1), &table(in, cpu_y_part, 0),
      0, NULL, NULL);
27   check (err, "Failed to read buffer from GPU");
28   ssandPile_refresh_img ();
29 }
30
31 unsigned ssandPile_invoke_ocl_ssand_hybrid (unsigned nb_iter) {
32   size_t global[2] = {DIM, gpu_y_part}; // global domain size for our calculation
33   size_t local[2]  = {GPU_TILE_W, GPU_TILE_H}; // local domain size for our
      calculation
34
35   cl_int err;
36   cl_event kernel_event;
37
38   int diff = 0;
39   for (unsigned it = 1; it <= nb_iter; it++) {
40     ++itVal;
41
42     // Set kernel arguments
```

```
43      err = 0;
44      ...
45      err |= clSetKernelArg (compute_kernel, 3, sizeof (unsigned), &cpu_y_part);
46      err |= clSetKernelArg (compute_kernel, 4, sizeof (unsigned int), &itVal);
47      check (err, "Failed to set kernel arguments");
48
49      // Launch GPU kernel
50      ...
51      clFlush (queue);
52
53      int change = 0;
54  #pragma omp parallel for collapse(2) reduction(|:change) schedule(runtime)
55      for (int y = 0; y < cpu_y_part; y += TILE_H)
56        for (int x = 0; x < DIM; x += TILE_W)
57          change |=
58              do_tile(x + (x == 0), y + (y == 0), TILE_W - ((x + TILE_W == DIM) +
59                  (x == 0)), TILE_H - (y == 0), omp_get_thread_num() );
60
61      // // Read the first line processed by the GPU for CPU processing
62      err = clEnqueueReadBuffer (queue, next_buffer, CL_TRUE, cpu_y_part * DIM *
63          sizeof (unsigned), sizeof (unsigned) * DIM, &table(out, cpu_y_part, 0), 0,
64          NULL, NULL);
65      check (err, "Failed to read to buffer");
66
67      // Write the last line processed by the CPU for GPU processing
68      err = clEnqueueWriteBuffer (queue, next_buffer, CL_TRUE, (cpu_y_part-1) * DIM *
69          sizeof (unsigned), sizeof (unsigned) * (DIM), &table(out, (cpu_y_part-1), 0)
70          , 0, NULL, NULL);
71      check (err, "Failed to write to buffer");
72
73      { // Swap buffers
74        cl_mem tmp  = cur_buffer;
75        cur_buffer  = next_buffer;
76        next_buffer = tmp;
77      }
78      swap_tables();
79
80      ocl_monitor(kernel_event,0,cpu_y_part,global[0],global[1],TASK_TYPE_COMPUTE);
81      clReleaseEvent (kernel_event);
82
83      if(itVal > MAX_ITERATIONS) {
84        if(++percent == THIRTY_PERCENT) {
85          // manage terminate
86        }
87      }
88
89      clFinish (queue);
90    }
```

```
91
92   if (do_display) {
93     // Send CPU contribution to GPU memory
94     err = clEnqueueWriteBuffer(queue, cur_buffer, CL_TRUE, 0, DIM * (cpu_y_part-1)
95         * sizeof (unsigned), &table(in, 0, 0), 0, NULL, NULL);
96     check (err, "Failed to write to buffer");
97   }
98   return diff;
99 }
```

Listing 3 – Implémentation openCL + OpenMP "sandPile.c"

1.2 Présentation des expériences

L'ensemble des expériences que je présenterai ci-dessous est fait sur la première version avec une répartition 50/50 entre le CPU et le GPU. Cette première partie d'expériences est faite dans la salle 008 au CREMI avec la machine `miro`

1.2.1 Avec l'optimisation du rendement du pipeline(-wt opt)

Étant toujours fidèle aux meilleures distributions obtenues lors des précédents rendus, ces expériences sont faites sur la distribution `static` et `dynamic,4` sur une image de taille(512) en limitant le nombre d'itérations à 100. Sur ces deux distributions au delà de 22 threads on constate de plus en plus de bruitages avec un meilleur *speedup* conserver par la distribution `static` avec une tuile(16x16).

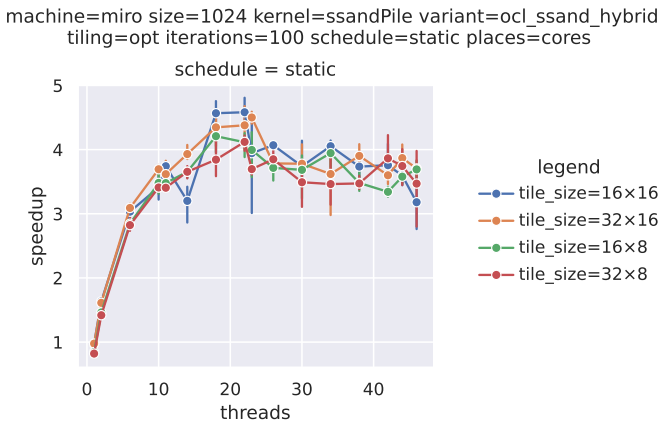

FIGURE 1 – *Expérience avec* `static` *pour différentes tailles de tuiles*

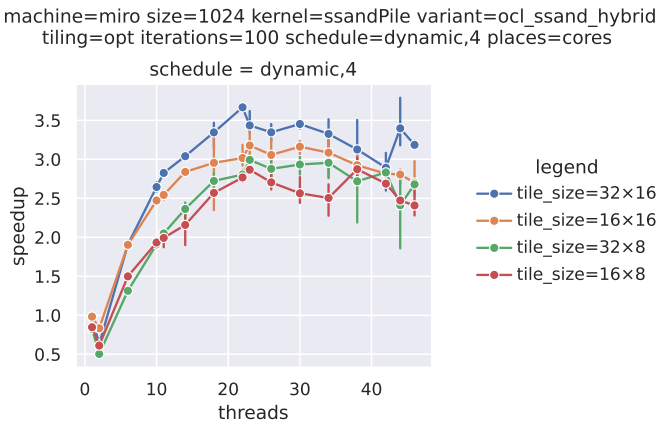

FIGURE 2 – *Expérience avec* `dynami,4` *pour différentes tailles de tuiles*

Sur la précédente expérience, certains points présentent un meilleur speedup mais vue le nombre d'itérations est petit, j'ai décidé de faire une jolie carte de chaleur sur chacun de ces points sur 1000 itérations avec une image(1024) dans le but de déterminer la meilleure répartition des tuiles applicables à cette distribution. Sachant que la largeur multipliée par la hauteur d'une tuile ne doit pas dépasser la dimension de l'image pour le traitement du GPU, j'ai donc fait en sorte de respecter cette contrainte. Après analyse, il en résulte que 18, 22, 23 et 44 threads présentent un meilleur speedup(9).

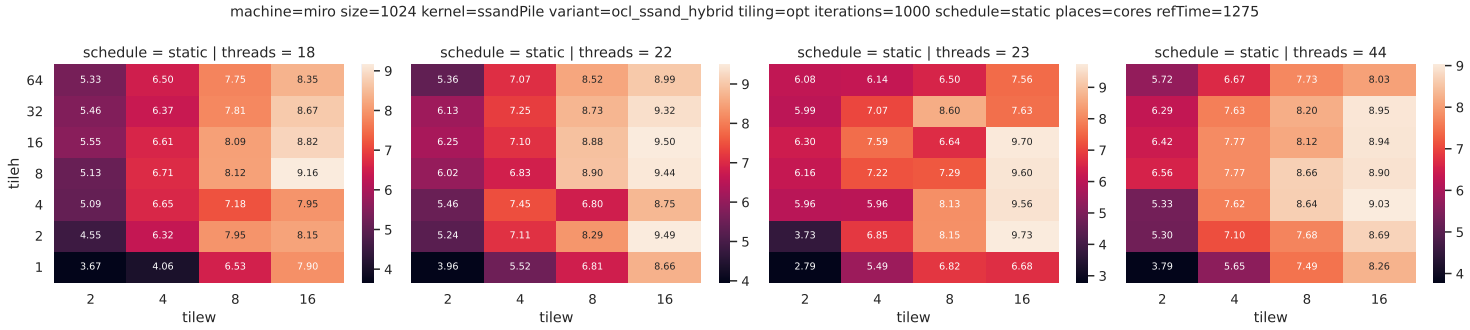

FIGURE 3 – *Carte de chaleur du speedup (n=9) pour la version* **`hybrid`** *en fonction de la taille des tuiles au voisinage du nombre de thread aux performances maximales.*

Vous trouverez ci-dessous une courbe jusqu'à stabilisation avec une dimension(512) et une tuile 16x16. on voit que ici qu'avec 22 threads on obtient un meilleur speedup ce qui est quasiment conforme aux données des cartes de chaleurs ci-dessus.

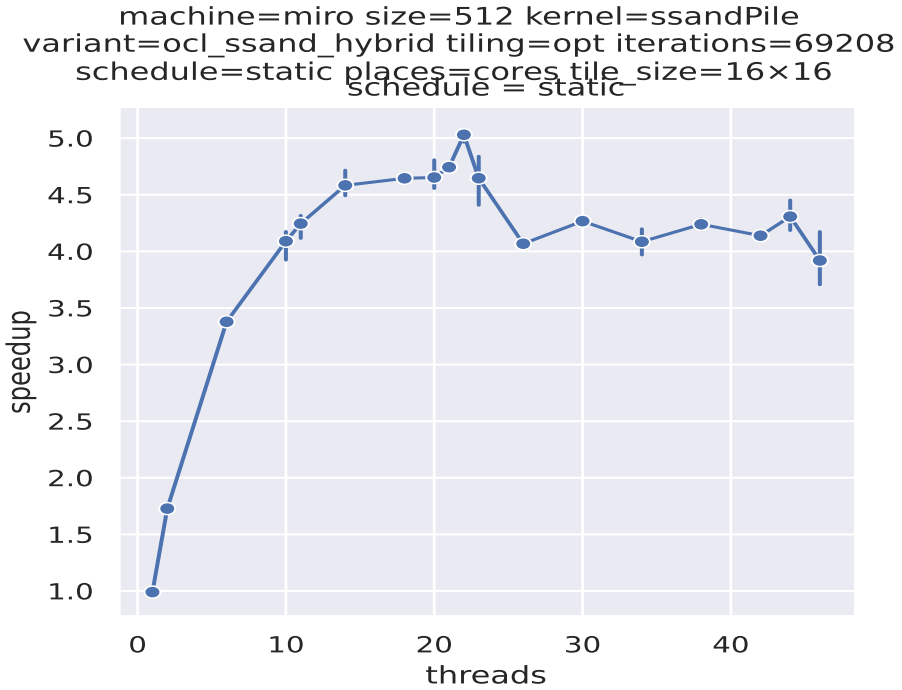

FIGURE 4 – *Expérience complète avec* **`schedule static`** *pour tileh=16 et tilew=16*

Une dernière courbe présentant les speedup pouvant être atteints selon la taille de l'image utilisée, on voit ici que la dimension(2048) est largement devant suivi de celle de 1024. Cette dernière est faite en salle(203) avec la machine `quark`.

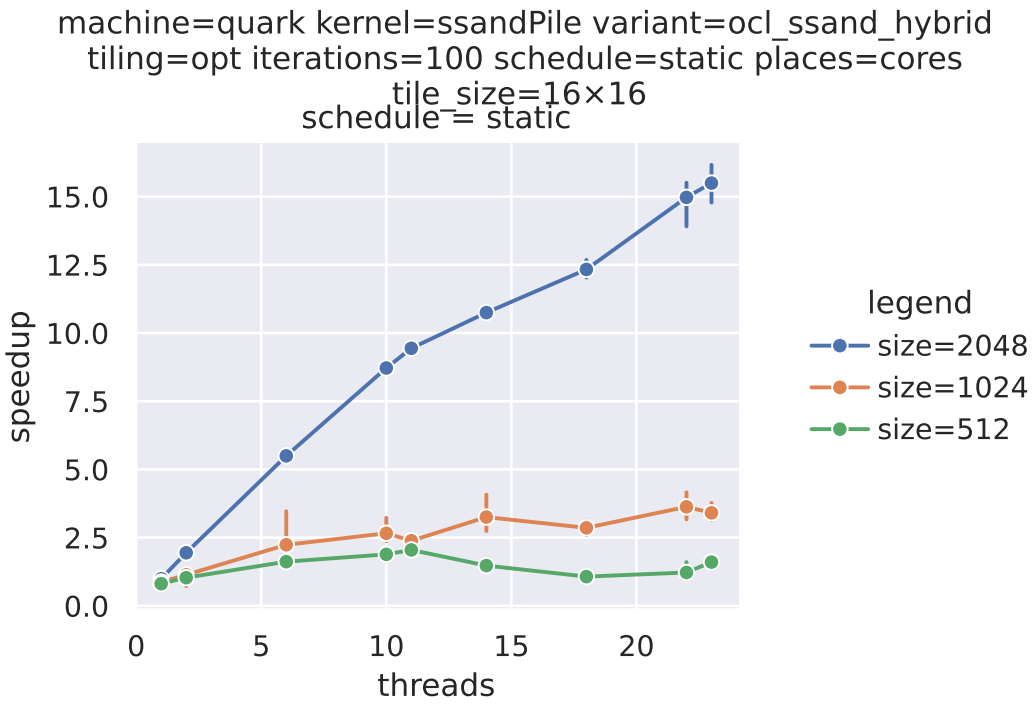

FIGURE 5 – *Expérience complète avec* `schedule static` *pour tileh=16 et tilew=16*

1.2.2 Comparaison entre l'optimisation du rendement de pipeline(opt) et la version vectorielle(avx) de l'OpenCL + CPU

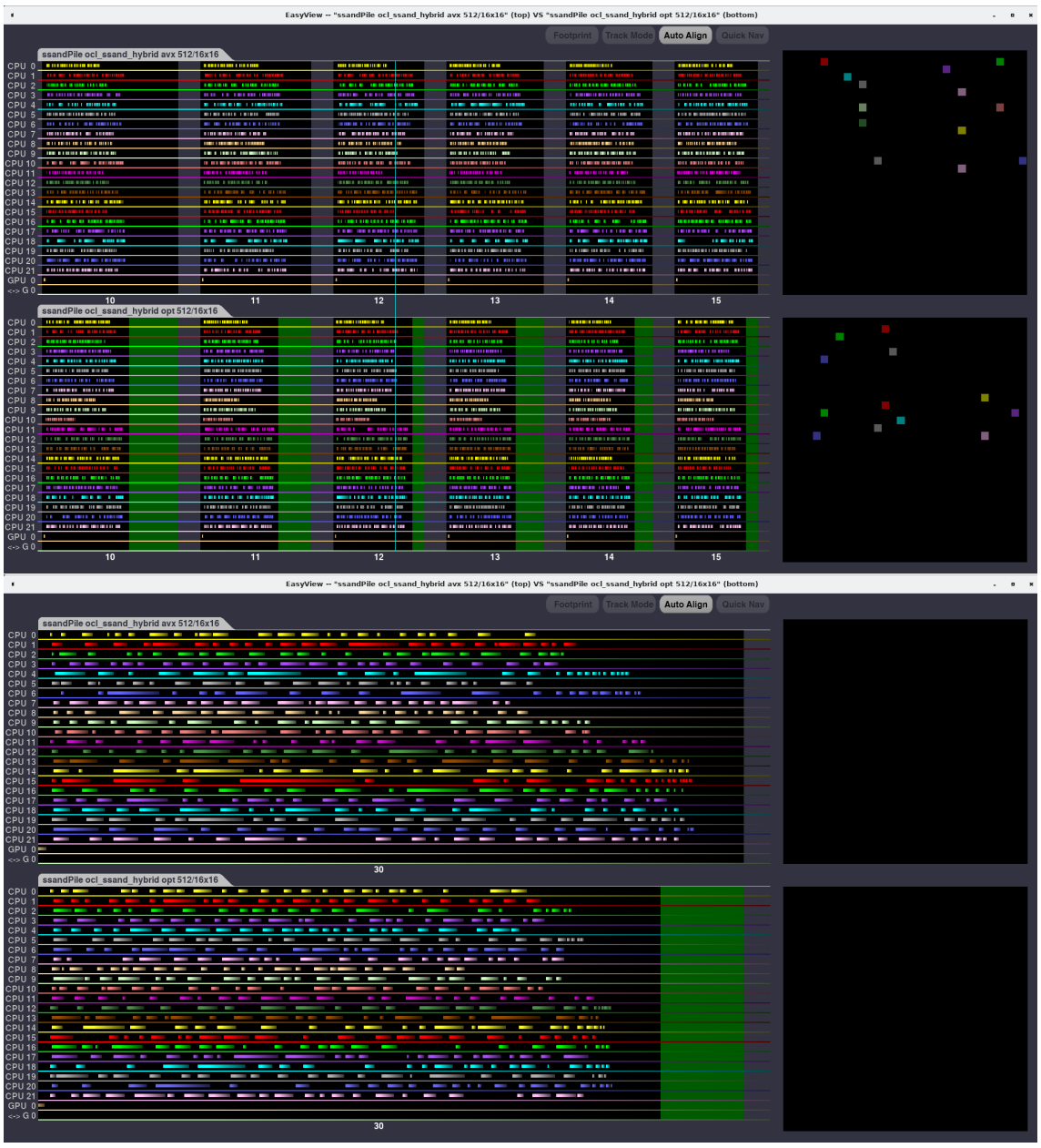

FIGURE 6 – *Traces comparative du calcul pour les premières itérations entre la version vectorielle(avx) et l'optimisation du pipelin(opt) avec leurs paramètres optimaux*

1.2.3 Comparaison entre l'OpenCL + CPU et OpenCL + GPU dynamique avec l'optimisation du rendement de pipeline(opt)

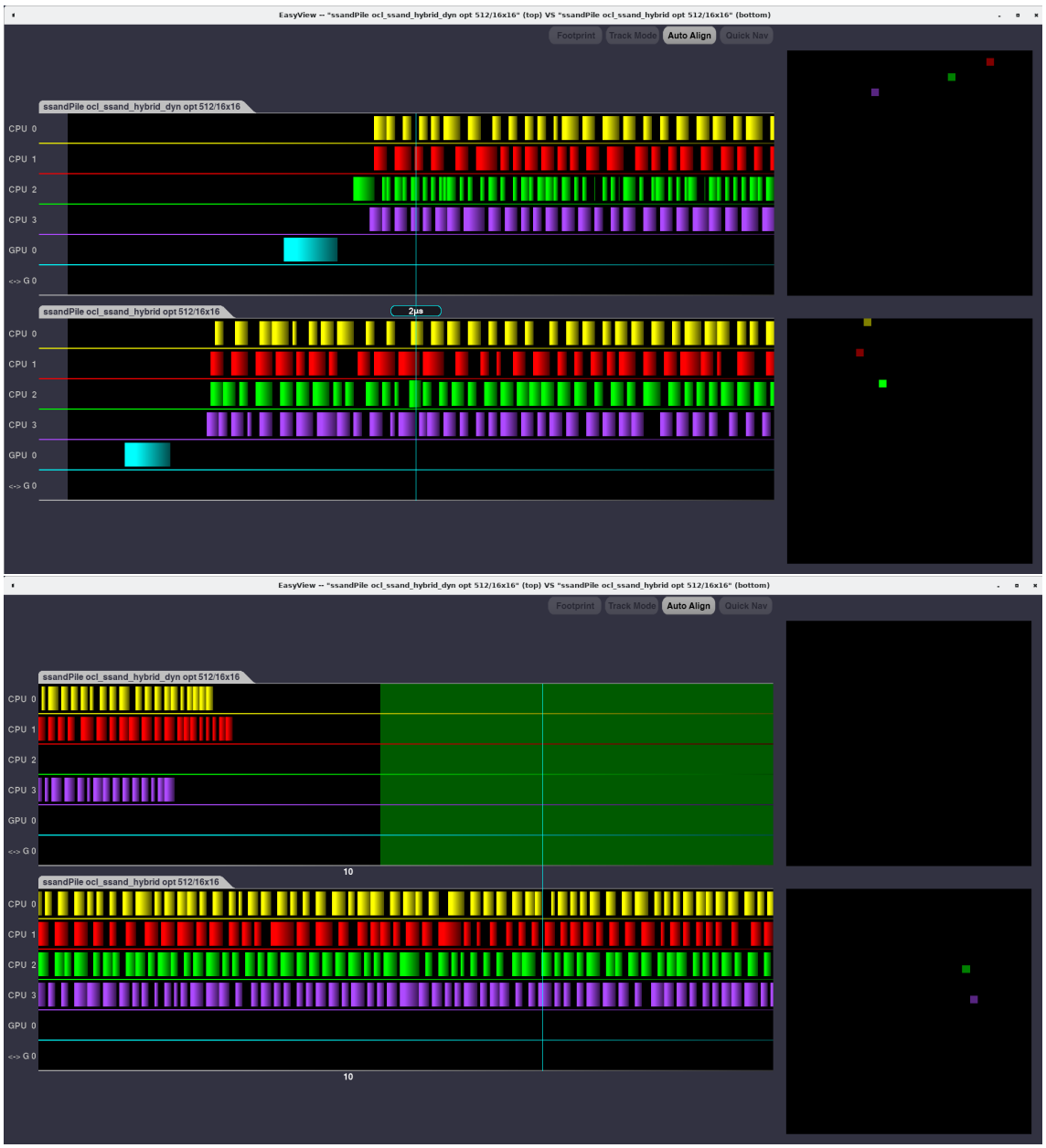

FIGURE 7 – *Traces comparative du calcul pour une itération entre la version OpenCL + CPU et OpenCL + GPU dynamique avec(-opt)*