Programmation des architectures Parallèles

Lucas Marques

May 6, 2025

Rapport Projet: Le jeu de la vie - Quatrième Jallon, OpenMP + OpenCL

1 Introduction

Avant toute chose, les expériences ont été réalisées sur le pc westmale de la salle 007 car je n'ai réussi à réveiller aucun pc de 008 ou 009, et n'ai pas pu me rendre physiquement au CREMI.
Voyons maintenant ce que l'on peut tirer de la combinaison GPU/CPU !

2 Première variante

2.1 Idée générale

Pour la première variante, j'ai souhaité aller au plus simple. L'objectif est de diviser la charge de travail et de la répartir entre CPU et GPU, et pour cette version j'aimerai avoir quelque-chose de proche de ce que nous avions abordés en cours, c'est-à-dire une version répartissant la charge entre CPU et GPU de manière statique (j'entend par là qu'on n'adapte pas la taille des zones de calcul au cours de l'exécution), avec bordure afin de permettre de ne faire la synchronisation CPU-GPU que toutes les n itérations, n. Je peux donc définir une fréquence de synchronisation que nous appelerons $CPU\_GPU\_SYNC\_FREQ$ et de là en dériver une constante $BORDER\_SIZE$ qui sera tout simplement égale à $CPU\_GPU\_SYNC\_FREQ$.
Je me doute que la partie la plus contraignante sera la partie GPU. En effet, j'aimerai tirer le plein potentiel du GPU et donc ne pas lancer $DIM * DIM$ threads pour n'en utiliser par exemple que $DIM * (512 + BORDER\_SIZE)$ si je choisis une taille de calcul sur le GPU de 512 lignes. Il faudra donc prendre la bordure dans la partie du domaine calculée par le GPU. Le GPU calculera donc en réalité $NB\_LINES\_FOR\_GPU - BORDER\_SIZE$, avec $NB\_LINES\_FOR\_GPU$ une puissance de 2 (inférieure à DIM bien entendu...).
Niveau synchronisation, le plus simple est de donner au GPU un domaine de calcul partant de la première ligne. Cela évitera des calculs superflus d'indice (et donc quelques heures de debug supplémentaires). Lorsque l'on synchronisera le GPU avec le CPU, on pourra utiliser $clEnqueueReadBuffer$ en donnant directement l'adresse de *_table* et la taille calculée par le GPU prenant en compte la bordure. Pour envoyer le nouvel état de la bordure du CPU au GPU, on pourra tout simplement utiliser clEnqueueWriteBuffer avec comme offset le nombre d'octets lu du GPU dans l'étape précédente. Cela nous permettra de ne transmettre sur les bus que les données essentielles ! Évidemment, le CPU se chargera de calculer la bordure et donc devra lui même avoir une bordure au dessus de la bordure du GPU.

Nous travaillerons également avec des char pour plus d'efficacité dans les transferts de donnée !
J'étais parti d'une première version un peu bricolée avant d'avoir les consignes complètes. Elle diffère très peu, à part qu'elle tentait d'utiliser un tuilage différent pour le CPU et le GPU (en trichant et en enlevant de main.c la vérification du tuilage pour le gpu...) mais je ne prendrai pas le temps de la tester car c'est un peu de la triche, et qu'accessoirement le temps manque grandement. Elle est disponible dans le code et utilisable si l'on définit la constante de préprocesseur TASKV. J'ai donc prit le temps de la retravailler (en comprenant notamment que clEnqueueNdRangeKernel n'était pas bloquant par défaut...) pour en arriver à cette version :

2.2 Implémentation

```
static inline ocl_sync_borders (cl_int err)
{
  unsigned true_gpu_size =
      sizeof (cell_t) * DIM * (kernel_fp[0].h - BORDER_SIZE);

  err = clEnqueueReadBuffer (
      ocl_queue (0), ocl_cur_buffer (0), CL_TRUE,
      sizeof (cell_t) * DIM * (kernel_fp[0].h - BORDER_SIZE * 2),
      sizeof (cell_t) * DIM * BORDER_SIZE,
      _table + DIM * (kernel_fp[0].h - BORDER_SIZE * 2), 0, NULL,
          NULL);
  check (err, "Err syncing host to device");

  size_t border_offset_elements = DIM * (kernel_fp[0].h -
      BORDER_SIZE);

  err =
      clEnqueueWriteBuffer (ocl_queue (0), ocl_cur_buffer (0),
          CL_TRUE,
                            true_gpu_size, BORDER_SIZE * DIM *
                                sizeof (cell_t),
                            _table + border_offset_elements, 0,
                                NULL, NULL);
  check (err, "Err syncing device to host");
}

unsigned life_omp_ocl_compute_ocl (unsigned nb_iter)
{
  size_t global[2] = {DIM,
                      kernel_fp[0].h}; // global domain size for
                          our calculation
  size_t local[2]  = {TILE_W, TILE_H}; // local domain size for our
       calculation
  cl_int err;
  uint64_t clock;
  unsigned change = 0;

  for (unsigned iter = 1; iter <= nb_iter; iter++) {
    enqueue_kernel (err, global, local, &clock);
    compute_cpu (&change);
    finish_and_time (clock);
    ocl_swap_tables ();
    if (++true_iter_number % GPU_CPU_SYNC_FREQ == 0 &&
        true_iter_number > 0)
      ocl_sync_borders (err);
  }
  return 0;
}
```

Je passe sur le code de *enqueue_kernel*, *compute_cpu*, *finish_and_time* et *ocl_swap_tables* qui sont triviaux, et évidemment disponibles dans le dépot git. *ocl_sync_borders* se charge simplement de synchroniser les bords larges calculés sur le CPU avec le GPU.

2.3 Performances

Nous ferons nos comparaisons avec la version ompfor, en utilisant comme fonction de tuile par défaut le code de la version opt d'une étape précédente.

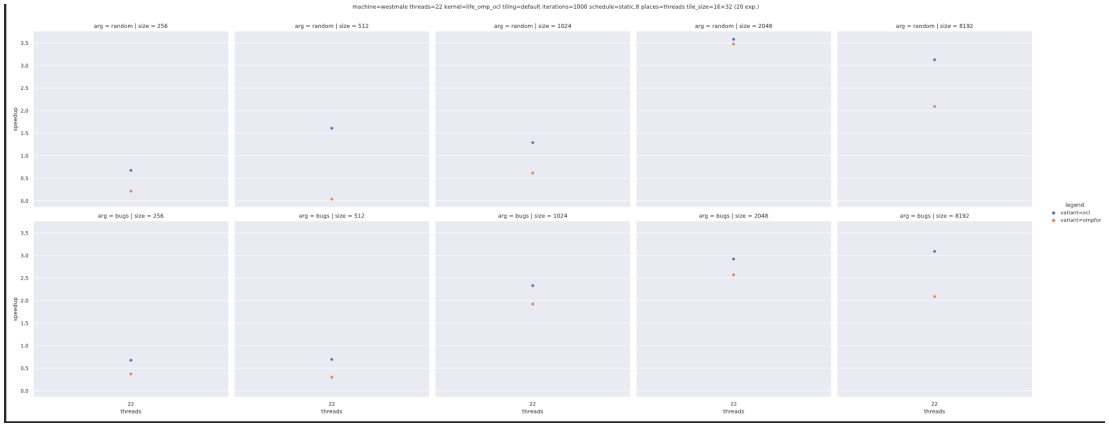

FIGURE 1 – ompfor vs ocl

On constate dors et déjà de légers speedups, et ce peu importe la version (logique, nous ne sommes pas en lazy...) et la taille. C'est plutôt bon signe, mais en observant des traces, je constate une distribution un peu cahotique des threads :

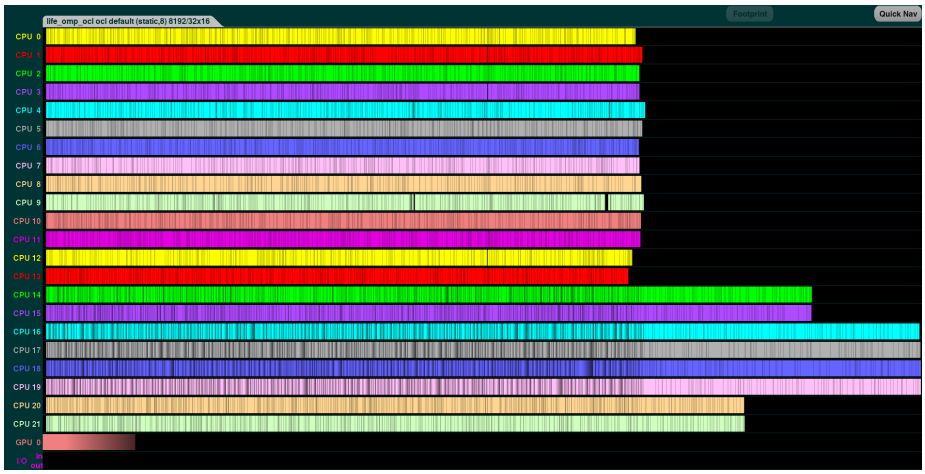

FIGURE 2 – Distribution threads static,8 classique pour ocl

Je ne suis pas sûr et certain de l'explication, je pense que cela a a voir avec la distribution initiale des threads et le fait que les tuiles ne mettent pas forcément le même temps, avec vraisemblablement pas mal de défauts de page même si j'ai essayé d'adapter le first touch (et de le forcer à être utilisé, il ne semblait pas être appelé quand lancé avec -g). Ce n'est cependant pas le sujet de ce rapport, nous n'allons donc pas chercher en profondeur leur cause. En essayant toute sorte de distribution plus ou moins farfelues, je me suis rendu compte que de manière générale, les distributions dynamiques s'en sortaient mieux.

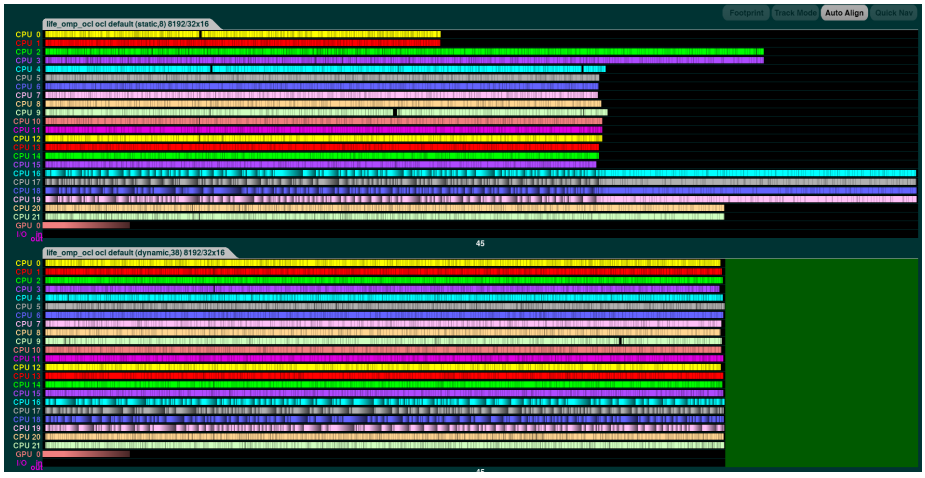

FIGURE 3 – Comparaison schedule statique (en bas) et dynamique (en haut)

On constate une meilleur répartition des tâches menant à beaucoup moins

de temps morts et donc une exécution plus efficace.

2.4 Impact des bords larges

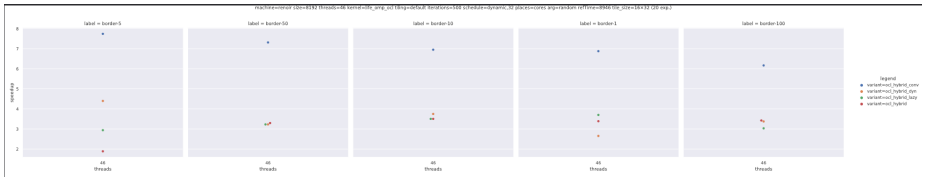

FIGURE 4 – Vitesse en fonction de la taille du bord large

Sur ma version, je constate qu'un bord de 5 donne déjà d'excellents résultats. C'est un peu étonnant, car un bord de 50 performe moins bien alors qu'on pourrait s'attendre au contraire.

3 Adaptation à la charge de travail

3.1 Idée générale

Fort de cette première version, j'ai donc souhaité faire une version avec la frontière qui se déplace selon la charge sur le CPU et le GPU, afin d'équilibrer au mieux les deux, car on constate dans les traces que la partie sur le GPU prend bien moins de temps que celle sur CPU. Je suis donc parti de la version de M. Namyst pour Mandel (qui m'avait déjà servi de base).
Pour cela, j'additione le temps prit par les deux kernel entre chaque synchronisation (afin d'éviter d'avoir à trop réfléchir sur comment synchroniser les bords d'une version ayant des bords pouvant se déplacer...), et si une version va substantiellement plus vite (j'ai choisi la valeur de 4x plus vite car elle semblait assez équilibrée, ne réattribuant pas trop souvent la charge), on lui rajoute une tuile en hauteur. J'ai un peu triché : si on donne plus de travail au GPU, on ne lui envoie qu'une tuile supplémentaire, par contre si c'est le CPU, on resynchronise toute la partie GPU. Je n'ai pour être parfaitement honnête pas réussi à n'envoyer qu'une seule tuile, et ne voulant pas perdre trop de temps pour avancer sur une meilleur version, j'ai gardé ce comportement.

3.2 Implémentation

```
unsigned life_omp_ocl_compute_ocl_adaptive (unsigned nb_iter)
{
  size_t global[2] = {DIM, kernel_fp[0].h};
  size_t local[2]  = {TILE_W, TILE_H};
  cl_int err;
  uint64_t clock;
  unsigned change = 0;

  for (unsigned iter = 1; iter <= nb_iter; iter++) {
    // gpu
    enqueue_kernel (err, global, local, &clock);
    // cpu
    compute_cpu (&change);
    finish_and_time_additive (clock);
    ocl_swap_tables ();
    if (++true_iter_number % GPU_CPU_SYNC_FREQ == 0 &&
        true_iter_number > 0) {
      ocl_sync_borders (err);
      if (kernel_durations[0]) {
        if (much_greater_than (kernel_durations[0],
                               kernel_durations[1]) &&
            kernel_fp[0].h > TILE_H * 2) { // cpu going faster
          PRINT_DEBUG ('v', "Giving more work to the CPU\n");
          clEnqueueReadBuffer (
              ocl_queue (0), ocl_cur_buffer (0), CL_TRUE, 0,
              sizeof (cell_t) * DIM * kernel_fp[0].h, _table, 0,
                  NULL,
              NULL); // TODO: make this load only whats required
          kernel_fp[0].h -= TILE_H;
          kernel_fp[1].h += TILE_H;
          kernel_fp[1].y = kernel_fp[0].y + kernel_fp[0].h;
          global[1]      = kernel_fp[0].h;
        } else if (much_greater_than (kernel_durations[1],
                                      kernel_durations[0]) &&
                   kernel_fp[1].h > TILE_H) { // gpu going brrrr
                       fast
          PRINT_DEBUG ('v', "Giving more work to the GPU\n");
          clEnqueueWriteBuffer (ocl_queue (0), ocl_cur_buffer (0),
              CL_TRUE,
                                sizeof (cell_t) * kernel_fp[0].h *
                                    DIM,
                                sizeof (cell_t) * DIM * TILE_H,
                                _table + (DIM * kernel_fp[0].h), 0,
                                     NULL, NULL);
          kernel_fp[0].h += TILE_H;
          kernel_fp[1].h -= TILE_H;
          kernel_fp[1].y = kernel_fp[0].y + kernel_fp[0].h;
          global[1]      = kernel_fp[0].h;
        } else {
          kernel_durations[0] = 0;
          kernel_durations[1] = 0;
        }
      }
    }
  }
  return 0;
}
```

On pourrait se dire que d'agrandir d'une seule tuile est un peu sous optimal, mais mon objectif était d'avoir une version basique fonctionnelle pour pouvoir ensuite venir avec une version convergeant plus vite, nous nous contenterons donc de cela pour l'instant.

3.3 Performances

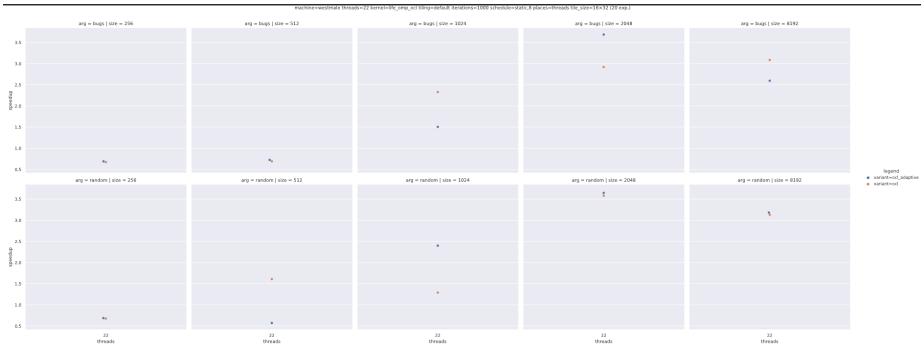

FIGURE 5 – OCL basique vs frontière dynamique simpliste

On semble constater des différences notables, mais comme prévu, avancer d'une tuile à la fois n'est peut-être pas optimal. Les traces ne sont pas beaucoup plus intéssantes, passons directement à la suite...

4 Convergence des frontières (et des luttes ?)

4.1 Idée générale

Plutôt que d'ajouter une tuile à la version la plus performantes, nous pourrions essayer de calculer un ratio de la différence du temps d'exécution des deux kernels, pour donner au kernel le plus rapide un nombre de tuile linéairement lié à ce dernier. Pour cela, on détermine quel kernel va le plus vite, on calcul ce dit ratio, on s'assure qu'il ne dépasse pas l'agrandissement maximum qui serait la taille du kernel le plus lent (modulo notre bordure afin de ne pas avoir à trop réfléchir encore une fois... le temps est compté !), et on agrandit ce côté (en trichant toujours de la même manière côté CPU).

4.2 Implémentation

```
unsigned life_omp_ocl_compute_ocl_adaptive_conv (unsigned nb_iter)
{
  size_t global[2] = {DIM, kernel_fp[0].h};
  size_t local[2]  = {TILE_W, TILE_H};
  cl_int err;
  uint64_t clock;
  unsigned change  = 0;
  int border_tiles = (BORDER_SIZE * 2) / TILE_H + 1;

  for (unsigned iter = 1; iter <= nb_iter; iter++) {
    // gpu
    enqueue_kernel (err, global, local, &clock);
    // cpu
    compute_cpu (&change);
    finish_and_time_additive (clock);
    ocl_swap_tables ();
    if (++true_iter_number % GPU_CPU_SYNC_FREQ == 0 &&
        true_iter_number > 0) {
      ocl_sync_borders (err);
      if (kernel_durations[0]) {
        // we will first look at which kernel is going the faster
        // to define the kernel that grows and the one that shrinks
            based
        unsigned growing_idx, shrinking_idx;
        if (much_greater_than (kernel_durations[0],
            kernel_durations[1])) {
          growing_idx   = 1;
          shrinking_idx = 0;
        } else if (much_greater_than (kernel_durations[1],
                                      kernel_durations[0])) {
          growing_idx   = 0;
          shrinking_idx = 1;
        } else {
          PRINT_DEBUG ('v', "skipping␣growth,␣%d␣-␣%d\n",
              kernel_durations[0],
                       kernel_durations[1]);
          goto skip_growth; // no one saw this .......
        }
        // now we compute both the maximal boundary grow and one
            that
        // matches the ratio cpu/gpu, we get the MIN of those two
            values
        unsigned max_growth =
            (kernel_fp[shrinking_idx].h - border_tiles * TILE_H) /
                TILE_H;
        unsigned ratio =
            kernel_durations[shrinking_idx] / kernel_durations[
                growing_idx];
        unsigned ratio_growth = (ratio * kernel_fp[growing_idx].h)
            / TILE_H;
        unsigned growth       = MIN (max_growth, ratio_growth);

        PRINT_DEBUG ('v', "growing␣%s␣by␣%d␣tiles␣(%d)\n",
                     growing_idx == 0 ? "device" : "host", growth,
                     growth * TILE_H);

        // first we sync, then we grow
        if (growing_idx == 1) {
          // growing on CPU, need to sync from device
          clEnqueueReadBuffer (ocl_queue (0), ocl_cur_buffer (0),
              CL_TRUE, 0,
                               sizeof (cell_t) * DIM * kernel_fp
                                   [0].h, _table,
                               0, NULL, NULL);
        } else {
          // growing on GPU, need to sync from host
          clEnqueueWriteBuffer (ocl_queue (0), ocl_cur_buffer (0),
              CL_TRUE,
```

```
57                                      sizeof (cell_t) * kernel_fp[0].h *
                                            DIM,
                                        sizeof (cell_t) * DIM * growth *
                                            TILE_H,
59                                      _table + (DIM * kernel_fp[0].h), 0,
                                             NULL, NULL);
          }
61
          kernel_fp[growing_idx].h += growth * TILE_H;
63        kernel_fp[shrinking_idx].h -= growth * TILE_H;
          kernel_fp[1].y = kernel_fp[0].y + kernel_fp[0].h;
65        global[1]      = kernel_fp[0].h;
        skip_growth:;
67      }
        kernel_durations[0] = 0;
69      kernel_durations[1] = 0;
      }
71  }
  return 0;
73 }
```

L'implémentation est à mes yeux un peu plus propre, malgré la présence d'un goto que je me permet d'utiliser car il évite juste un niveau d'indentation supplémentaire. Elle fait la même chose mais charge plus de tuiles. Voyons voir si cela suffit à avoir une différence significative !

4.3 Performances

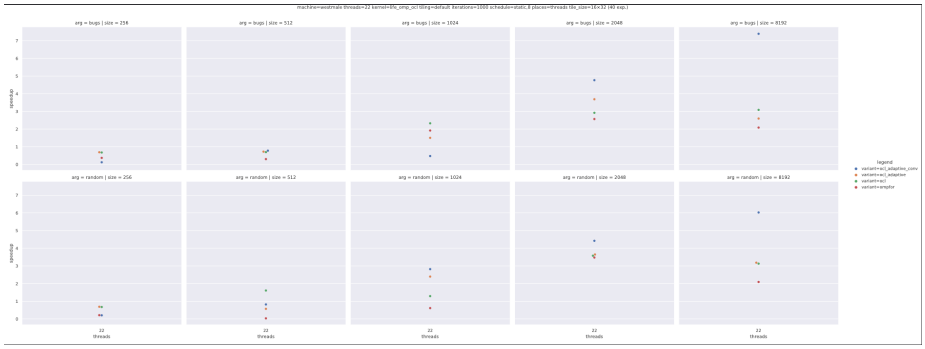

FIGURE 6 – Ajout de la version présentée précédemment aux benchmarks

Les différences sont nettement significatives, particulièrement sur une configuration de grande taille. C'est assez logique, on essaie de réduire la différence entre les deux versions, et lorsque l'on compare les traces des deux versions, c'est flagrant.

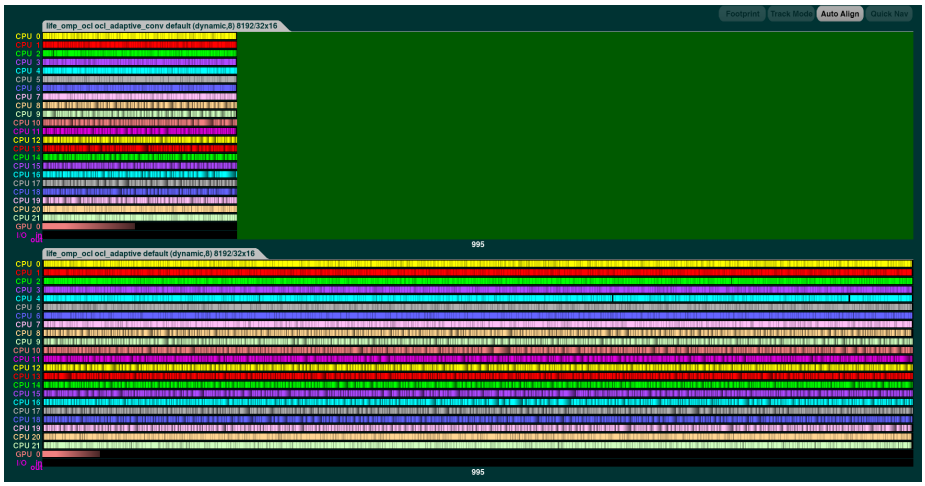

FIGURE 7 – Trace version adaptive de base vs avec ratio, beaucoup d'itérations

Le temps d'une exécution à un nombre d'itération avancé est nettement inférieur pour la version sensée converger plus rapidement. Le ratio GPU/CPU est également largement différent, on voit que le GPU s'exécute (d'après les mesures, et l'incertitude induite par celles-ci) à peu près 2x plus vite que le CPU, alors qu'on était sur une exécution quasiment 10x plus rapide avant, et donc avions de gros temps morts où le GPU ne servait à rien. Et si l'on refait cette même comparaison au début de l'exécution, alors que l'on a pas encore eu le temps de converger vers un ratio plus optimal :

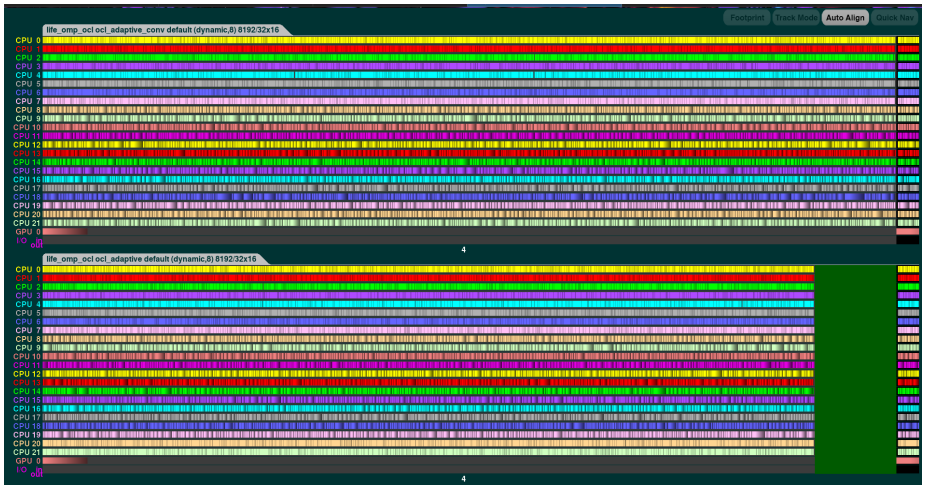

FIGURE 8 – Trace version adaptive de base vs avec ratio, peu d'itérations

On voit bien qu'on a deux traces beaucoup plus semblables, et cela met

bien en évidence l'impact du nombre d'itérations sur cette deuxième variante à frontière dynamique !
En bonus une petite vidéo que j'ai mit en privé sur Youtube qui montre bien le comportement : https://www.youtube.com/watch?v=iMXCqJXXDX0. Le petit icône dans la barre de tâche qui change au clic de ma souris me permet de choisir parmis trois CPU governors : la feuille verte = économie d'énergie, la balance violette = balanced et l'éclair rouge performance. On voit bien que quand je passe en économie d'énergie (qui est assez poussé en terme de limitation de cores/fréquence), le GPU prend rapidement le relais.

5 Version paresseuse hybride de base

5.1 Implémentation

```
unsigned life_omp_ocl_compute_ocl_hybrid_lazy (unsigned nb_iter)
{
  size_t global[2] = {DIM, kernel_fp[0].h};
  size_t local[2]  = {TILE_W, TILE_H};
  cl_int err;
  uint64_t clock;
  unsigned change = 0;

  for (unsigned iter = 1; iter <= nb_iter; iter++) {
    // computing GPU
    err = 0;
    err |= clSetKernelArg (ocl_compute_kernel (0), 0, sizeof (
        cl_mem),
                           &ocl_cur_buffer (0));
    err |= clSetKernelArg (ocl_compute_kernel (0), 1, sizeof (
        cl_mem),
                           &ocl_next_buffer (0));
    err |=
        clSetKernelArg (ocl_compute_kernel (0), 2, sizeof (cl_mem),
            &tile_in);
    err |=
        clSetKernelArg (ocl_compute_kernel (0), 3, sizeof (cl_mem),
            &tile_out);
    check (err, "Failed to set kernel computing arguments");

    clock = ezm_gettime ();
    clEnqueueNDRangeKernel (ocl_queue (0), ocl_compute_kernel (0),
        2, NULL,
                            global, local, 0, NULL,
                            ezp_ocl_eventptr (EVENT_START_KERNEL,
                                0));
    check (err, "Failed to execute kernel");
    int border_tiles = (BORDER_SIZE * 2) / TILE_H + 1;
    int cpu_start_y  = kernel_fp[0].h - (border_tiles * TILE_H);

    // computing CPU
#pragma omp parallel for reduction(| : change) collapse(2) schedule
    (runtime)
    for (int y = cpu_start_y; y < DIM; y += TILE_H) {
      for (int x = kernel_fp[1].x; x < DIM; x += TILE_W) {
        unsigned local_change = 0;
        unsigned tile_y       = y / TILE_H;
        unsigned tile_x       = x / TILE_W;
        // checking if we should recompute this tile or not
        if (cur_tiles (tile_y, tile_x)) {
          // we need to keep track of per-tile changes
          local_change = do_tile (x, y, TILE_W, TILE_H);
          change |= local_change;
```

```
43            if (local_change) {
                next_tiles (tile_y - 1, tile_x - 1) = 1;
45              next_tiles (tile_y - 1, tile_x)     = 1;
                next_tiles (tile_y - 1, tile_x + 1) = 1;
47              next_tiles (tile_y, tile_x - 1)     = 1;
                next_tiles (tile_y, tile_x)         = 1;
49              next_tiles (tile_y, tile_x + 1)     = 1;
                next_tiles (tile_y + 1, tile_x - 1) = 1;
51              next_tiles (tile_y + 1, tile_x)     = 1;
                next_tiles (tile_y + 1, tile_x + 1) = 1;
53            }
            }
55        }
      }
57    kernel_durations[1] += ezm_gettime () - clock;
      clFinish (ocl_queue (0));
59    kernel_durations[0] += ezp_gpu_event_monitor (
          0, EVENT_START_KERNEL, clock, &kernel_fp[0],
              TASK_TYPE_COMPUTE, 0);
61    ezp_gpu_event_reset ();

63    // switching tables
      {
65      cl_mem tmp_buf      = ocl_next_buffer (0);
        ocl_next_buffer (0) = ocl_cur_buffer (0);
67      ocl_cur_buffer (0)  = tmp_buf;
        tmp_buf             = tile_in;
69      tile_in             = tile_out;
        tile_out            = tmp_buf;
71
        cell_t *tmp  = _table;
73      cell_t *tmp2 = _tiles;

75      _table           = _alternate_table;
        _alternate_table = tmp;
77
        unsigned size = (DIM / TILE_W + 2) * (DIM / TILE_H + 2) *
            sizeof (cell_t);
79      _tiles           = _alternate_tiles;
        _alternate_tiles = tmp2;
81      memset (_alternate_tiles, 0, size);
      }
83
      if (++true_iter_number % BORDER_SIZE) {
85      unsigned true_gpu_size =
            sizeof (cell_t) * DIM * (kernel_fp[0].h - BORDER_SIZE);
87
        err = clEnqueueReadBuffer (
89          ocl_queue (0), ocl_cur_buffer (0), CL_TRUE,
            sizeof (cell_t) * DIM * (kernel_fp[0].h - BORDER_SIZE *
                2),
91          sizeof (cell_t) * DIM * BORDER_SIZE,
            _table + DIM * (kernel_fp[0].h - BORDER_SIZE * 2), 0,
                NULL, NULL);
93      check (err, "Err syncing host to device");

95      size_t border_offset_elements = DIM * (kernel_fp[0].h -
            BORDER_SIZE);

97      err = clEnqueueWriteBuffer (
            ocl_queue (0), ocl_cur_buffer (0), CL_TRUE, true_gpu_size
                ,
99          BORDER_SIZE * DIM * sizeof (cell_t), _table +
                border_offset_elements,
            0, NULL, NULL);
101     check (err, "Err syncing device to host");

103     if (kernel_durations[0]) {
          // we will first look at which kernel is going the faster
```

```
105         // to define the kernel that grows and the one that shrinks
                based
            unsigned growing_idx, shrinking_idx;
107         if (much_greater_than (kernel_durations[0],
                kernel_durations[1])) {
              growing_idx   = 1;
109           shrinking_idx = 0;
            } else if (much_greater_than (kernel_durations[1],
111                                       kernel_durations[0])) {
              growing_idx   = 0;
113           shrinking_idx = 1;
            } else {
115           PRINT_DEBUG ('v', "skipping␣growth,␣%d␣-␣%d\n",
                  kernel_durations[0],
                            kernel_durations[1]);
117           goto skip_growth; // no one saw this .......
            }
119         // now we compute both the maximal boundary grow and one
                that
            // matches the ratio cpu/gpu, we get the MIN of those two
                values
121         unsigned max_growth =
                (kernel_fp[shrinking_idx].h - border_tiles * TILE_H) /
                    TILE_H;
123         unsigned ratio =
                kernel_durations[shrinking_idx] / kernel_durations[
                    growing_idx];
125         unsigned ratio_growth = (ratio * kernel_fp[growing_idx].h)
                / TILE_H;
            unsigned growth       = MIN (max_growth, ratio_growth);
127
            if (growth == 0)
129           goto skip_growth;

131         PRINT_DEBUG ('v', "growing␣%s␣by␣%d␣tiles␣(%d)\n",
                         growing_idx == 0 ? "device" : "host", growth,
133                      growth * TILE_H);

135         // first we sync, then we grow
            if (growing_idx == 1) {
137           // growing on CPU, need to sync from device
              clEnqueueReadBuffer (ocl_queue (0), ocl_cur_buffer (0),
                  CL_TRUE, 0,
139                                sizeof (cell_t) * DIM * kernel_fp
                                    [0].h, _table,
                                0, NULL, NULL);
141         } else {
              // growing on GPU, need to sync from host
143           clEnqueueWriteBuffer (ocl_queue (0), ocl_cur_buffer (0),
                  CL_TRUE,
                                  sizeof (cell_t) * kernel_fp[0].h *
                                      DIM,
145                               sizeof (cell_t) * DIM * growth *
                                      TILE_H,
                                  _table + (DIM * kernel_fp[0].h), 0,
                                       NULL, NULL);
147         }

149         kernel_fp[growing_idx].h += growth * TILE_H;
            kernel_fp[shrinking_idx].h -= growth * TILE_H;
151         kernel_fp[1].y = kernel_fp[0].y + kernel_fp[0].h;
            global[1]      = kernel_fp[0].h;
153       skip_growth:;
          }
155     }
      }
157   return 0;
}
```

Tant qu'à faire, pourquoi ne pas essayer de prendre le temps d'adapter les ver-

sions paresseuses développées précédemment en kernel hybride. Je n'ai à l'heure où j'écris ce rapport pas réussi à faire une version hybride avec frontière dynamique et paresseuse, mais j'ai déjà pu faire une version hybride. Il faudrait synchroniser correctement les tuiles, en prenant en compte la bordure, le halo etc. et je n'ai pour l'instant pas réussi ! Mais cette version reste vraisemblablement la plus efficace que j'ai pu produire pour l'instant toute étape confondue...

6 Evaluations finale...

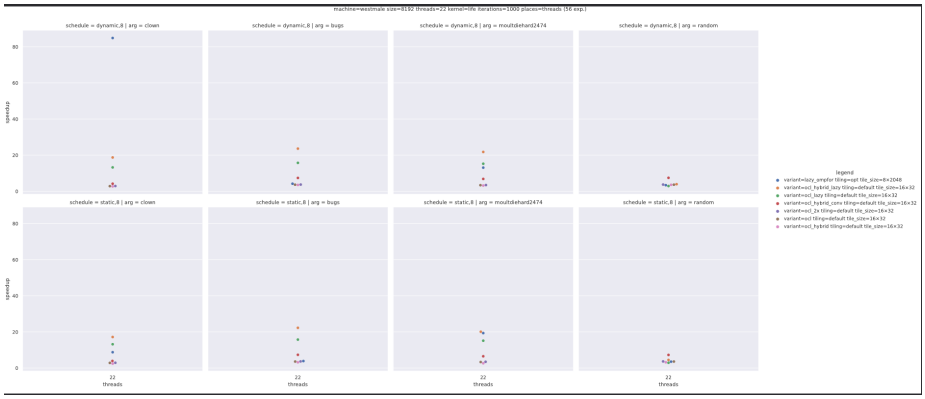

FIGURE 9 – Comparaison de plusieurs versions de différentes étapes

Comparons désormais toutes les versions intéressantes. Nous ne nous étendrons que sur celles faites dans cette étape même si j'en ai rajouté d'autres en comparaison.
A part pour clown, on constate bien l'efficacité de la version hybride lazy sur les variantes lazy-friendly... Pour clown, lazy_ompfor gagne probablement car le pattern est pile sur la démarcation, et l'overhead du kernel hybride peut difficilement rivaliser. Il faudrait une version à frontière dynamique ! Peut-être aurais-je le temps de la faire cette nuit ? ! (en reprenant le rapport le lendemain de sa première rédaction, je tiens à préciser que je n'aurai réussi à rien à part sacrifier la moitié de ma nuit de sommeil, mais c'est un challenge intéressant).

7 Conclusion

Bien que particulièrement perdu au début de l'UE, j'ai pu en travaillant (où plutôt en jouant la plupart du temps) avec easyPaP, tout en lisant à droite à gauche et en regardant quelques vidéos m'améliorer grandement dans la matière. J'aimais déjà beaucoup la programmation système, et les aspects nouveaux qui rentrent en compte dans la programmation parallèle me plaisent énormément. Je me suis beaucoup perdu à essayer de tester tout et n'importe quoi pour chercher

le meilleur speedup, pour ensuite essayer de le relier à mes connaissances, et je pense demander CISD l'année prochaine afin de pouvoir continuer sur cette voie !