Programmation Parallèle - Tas de sable abéliens

Etape 4 - OpenCL + OpenMP Implementation

Corentin Drezen

Mai 2024

1 Avant propos

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

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

- **sandPile_omp_ocl.h** : *OpenCL + OpenMP Implementation*

Comme je suis dans un trinôme je ne peux merge toute ma partie sur la branche principale car un autre élève fait la même partie. Mon code est donc dans la branche **gpu_omp_cdrezen**

2 Sending only one pixel line to the CPU and one to the GPU

Pour que le GPU calcule seulement sa partie, j'ai modifié le code OpenCL du rendu précédent pour pouvoir lui envoyer la position à partir de laquelle il commence à calculer..

```
__kernel void ssandPile_ocl_omp(__global unsigned *in, __global unsigned *out, unsigned border)
{
  int x = get_global_id (0);
  int y = get_global_id (1);
  int pos = y * DIM + x;

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

J'ai d'abord choisi de lancer le calcul de l'autre partie (côté CPU) à la suite du calcul GPU, donc les deux parties ne se calculent pas simultanément et elles peuvent être interdépendantes.

```
  ...
  for (unsigned it = 1; it <= nb_iter; it++) {
    _it++;

    //do_cpu_pre();

    // Set kernel arguments
    //
    err = 0;
    err |= clSetKernelArg (compute_kernel, 0, sizeof (cl_mem), &in_buff);
    err |= clSetKernelArg (compute_kernel, 1, sizeof (cl_mem), &out_buff);
    err |= clSetKernelArg (compute_kernel, 2, sizeof (unsigned), &current_border);
    check (err, "Failed to set kernel arguments");

    err = clEnqueueNDRangeKernel (queue, compute_kernel, 2, NULL, global, local, 0, NULL, NULL);
    check (err, "Failed to execute kernel");

    struct cpu_result result = do_cpu_post(_border, read_offset, read_ptr, read_sz,
                                           write_offset, write_y, write_sz);
    ...
```

Le calcul côté CPU et la lecture/écriture des lignes à la frontière s'est d'abord fait comme ci dessous.

```
    ...
    //read gpu line at border
    cl_int err = clEnqueueReadBuffer(queue, in_buff, CL_TRUE, read_offset,
                                     read_sz, read_ptr, 0, NULL, NULL);
    check(err, "Failed to read buffer from GPU");

      #pragma omp parallel for collapse(2) schedule(runtime)
    for(int y = 0; y < border + (!solo); y+=TILE_H) {
      for(int x = 0; x < DIM; x+=TILE_W)
      {
        bool 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)));
      }
    }

    //write cpu line(s) at border to gpu
    cl_int err = clEnqueueWriteBuffer (queue, out_buff, CL_TRUE, write_offset,
                                       write_sz, &table(out, write_y, 0), 0, NULL, NULL);
    check(err, "Failed to write buffer to GPU");
    nb_copied_lines = NB_LINES;

    swap_tables();
}
```

3 Less frequent communications

Pour réduire la fréquence des lectures, une condition où il n'est pas nécessaire d'envoyer de ligne au GPU était évidente, c'est le non-changement du bord. On a donc calculé le changement comme dans les *compute* habituels, et spécifiquement celui du bord de manière similaire, et conditionné l'écriture dans le buffer.

J'ai aussi essayé de faire calculer le changement de la ligne en bordure par le GPU, mais je n'ai pas réussi. Il faudrait le faire de manière atomique je pense avec le code actuel, car il traite beaucoup de cellules en parallèle d'après le fonctionnement d'un GPU.
Cela aurait probablement été contre productif pour réduit le nombre de communications de toute façon car je comptais lire cette information du GPU à chaque itération.

Je n'ai pas réussi à trouvé dans quelles autre conditions le calcul de l'un ne dépendait pas de l'autre mais il y en a sans doute une que j'ai manqué.
Cependant j'ai trouvé que l'on pouvait rendre le calcul autonome au moins pour une itération sur 2 mais je suis pas sûr d'avoir compris pourquoi.

Il m'a aussi semblé que réduire le nombre de données transmisse à chaque itération améliorait les performance avec mon code actuelle mais que limiter la 'synchronisation' du bord nécessitait de conserver plus de lignes. J'ai donc trouvé que 2 lignes copiés était optimal car je n'arrive pas à rendre les calculs vraiment autonomes à plus de de 2 itérations et qu'une ligne ne suffit pas.

```
struct cpu_result do_cpu_post(const unsigned border, const size_t read_offset, const TYPE* read_ptr, c
{
    unsigned nb_copied_lines = 0;

    if(!solo) //&& gpu_change[BORDER]
    {
      //read gpu line at border
      cl_int err = clEnqueueReadBuffer(queue, in_buff, CL_TRUE, read_offset,
                                       read_sz, read_ptr, 0, NULL, NULL);
      check(err, "Failed to read buffer from GPU");
    }

    bool change = 0;
    bool cpu_border_changed = false;

      #pragma omp parallel for collapse(2) schedule(runtime) reduction(|:change, cpu_border_changed)
    for(int y = 0; y < border + (!solo); y+=TILE_H) {
      for(int x = 0; x < DIM; x+=TILE_W)
      {
        bool 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)));

        if (y == border - TILE_H)
        {
```

```
          cpu_border_changed |= diff;
        }

        change |= diff;
      }
    }

    if(!solo && cpu_border_changed) //
    {
      //write cpu line(s) at border to gpu
      cl_int err = clEnqueueWriteBuffer (queue, out_buff, CL_TRUE, write_offset,
                                         write_sz, &table(out, write_y, 0), 0, NULL, NULL);
      check(err, "Failed to write buffer to GPU");
      nb_copied_lines = NB_LINES;
    }

    // Swap buffers
    {
      cl_mem tmp  = in_buff;
      in_buff = out_buff;
      out_buff = tmp;
      swap_tables();
      solo = !solo;
      //solo = (_it % NB_LINES);
    }

    return (struct cpu_result){ change, nb_copied_lines };
}
```

4 Optimisations

Comme le nombre de lignes copié au GPU peut maintenant varier, on met à jour la ligne à partir duquel il commence à calculer en conséquence.

On a aussi fait en sorte de que le compilateur reconnaisse que la plupart des variables utilisés dans *do_cpu_post* ne changerons pas après l'initialisation. Ce qui a beaucoup amélioré les performances.

Malheureusement cela ne me semble pas avoir suffit à rendre mon implémentation aussi rapide que le calcule 100% gpu ou 100% cpu. Elle est quand même dans de rare cas à peu près 2x plus rapide que ma meilleur implémention cpu mais significativement plus lente que la simple implementation 100% gpu sur une machine de la salle 008 avec une RTX 2070.

Je pense que c'est d'une part parce que il y a toujours peu d'autonomie entre les deux calculs et qu'il sont effectués à la suite donc un des deux est toujours inactif.

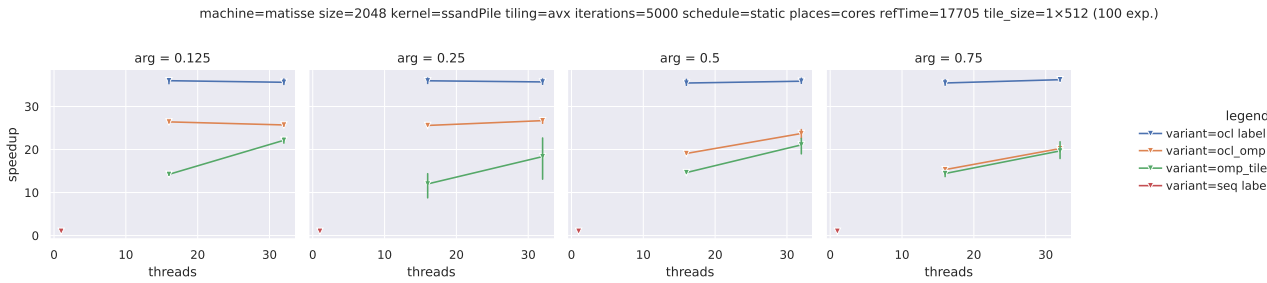

Figure 1: Experience qui met en avant ocl_omp en orange

5 CPU and GPU asynchronously

J'ai ensuite fait s'exécuter le CPU et GPU en parallèle et tenté une communication comme si dessous mais avec le nombre d'élément qui semblent nécessités d'être synchronisé cela ne me parait pas une bonne solution et je pense que l'un ralentirait l'autre.

J'ai pensé à une implémentions qui communiquerait quand un composant est en avance sur l'autre mais elle n'est pas encore au point.

```
bool do_cpu_parallel(unsigned it_cpu, const unsigned border, const size_t read_offset, const TYPE* read_ptr, cons
{
    unsigned nb_copied_lines = 0;

    if(it_cpu < _it)
    {
      //read gpu line at border
      cl_int err = clEnqueueReadBuffer(queue, in_buff, CL_TRUE, read_offset,
                                       read_sz, read_ptr, 0, NULL, NULL);
      check(err, "Failed to read buffer from GPU");
      _it = it_cpu;
    }

    bool change = 0;
    bool cpu_border_changed = false;

      #pragma omp parallel for collapse(2) schedule(runtime) reduction(|:change, cpu_border_changed)
    for(int y = 0; y < border + (it_cpu != _it); y+=TILE_H) {
      for(int x = 0; x < DIM; x+=TILE_W)
      {
        bool 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)));

        if (y == border - TILE_H)
        {
          cpu_border_changed |= diff;
        }
        change |= diff;
      }
    }
```

```
    if(cpu_border_changed && it_cpu > _it > 0)//
    {
      //write cpu line(s) at border to gpu
      cl_int err = clEnqueueWriteBuffer (queue, out_buff, CL_TRUE, write_offset,
                                         write_sz, &table(out, write_y, 0), 0, NULL, NULL);
      check(err, "Failed to write buffer to GPU");
      nb_copied_lines = NB_LINES;
      _it = it_cpu;
    }

    // Swap buffers
    {
      swap_tables();
    }
    return change;
}
```

Figure 2: avx512(appelée 'avx') vs avx256

6 Detecting termination & Generating dumps

Le *change* habituel côté cpu à simplement été utilisé pour detecter la terminaison de l'ensemble, cela semble fonctionner même pour des part cpu de 1/8, avec le *draw* par défault en tout cas. Cela permet aussi d'eviter de régulierement lire un buffer gpu de changement.

La génération du dump et l'affichage ont été crucial dans l'avancement du projet car aucun des deux ne fonctionnait alors que l'implémentation initial fonctionnait.

7 Load balancing

Il ne m'a pas semblé pertinent de faire de load balancing tant que le calcul 100% GPU ou 100% CPU est toujours plus rapide que l'hybride et que les deux composants ne sont pas constamment utilisés.

8 Argument de pourcentage cpu

J'ai fait en sorte qu'on puisse lancer *omp_ocl* avec un argument pour choisir la part cpu et ainsi faciliter les expériences. Toutes les fractions ne sont pas garanties de fonctionner car on fait des opérations sur des entiers et il y a aussi les dimensions de tuiles qui entrent en jeux.

J'ai du rajouté une prise en charge des arguments pour draw car j'utilisais l'argument -a pour modifier la part cpu. Pour utiliser en argument la part cpu et le draw en même temps il faut écrire quelque chose comme *-a 0.166spirals*

9 Bug argument draw

Il semble que *spirals* ne fonctionne qu'avec certains paramètres. C'est peut être du à l'alternation de l'autonomie (solo=!solo) qui fonctionne pour le draw par défault mais peut être pas pour le cas de spirals.
Peut être que plus de lignes devraient être transmises.
Changer la frontière cpu/gpu et utiliser le calcul de tuile 'avx' semble empêcher le bug.

10 Traces

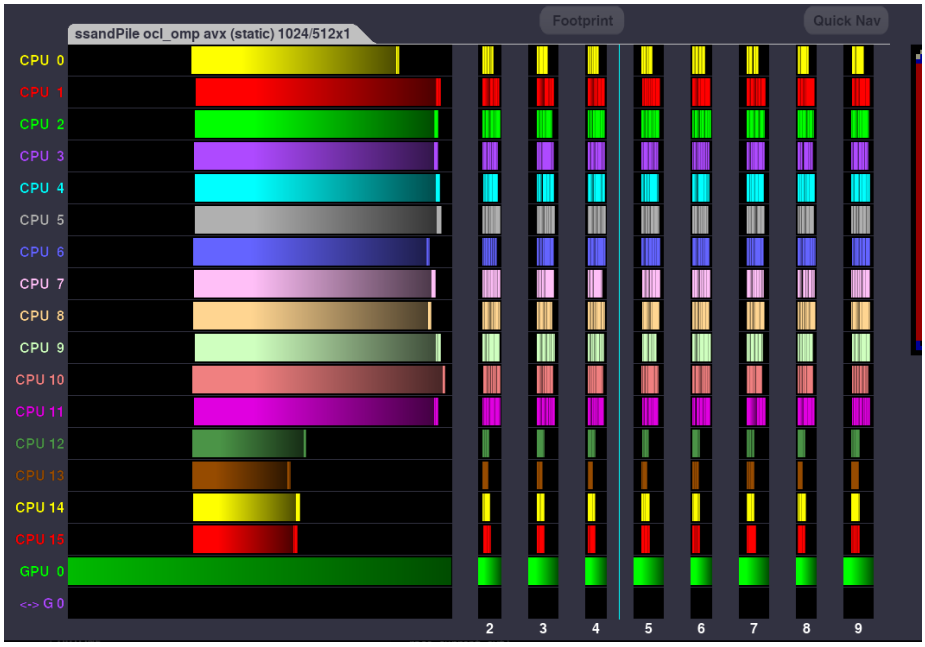

Figure 3: trace 1024

On peut voir que la première itération est coûteuse et que l'on peut se dispenser de 3 ou 4 thread.
Le gpu travailerait constament ou il n'est pas possible de bien le représenter dans les traces.

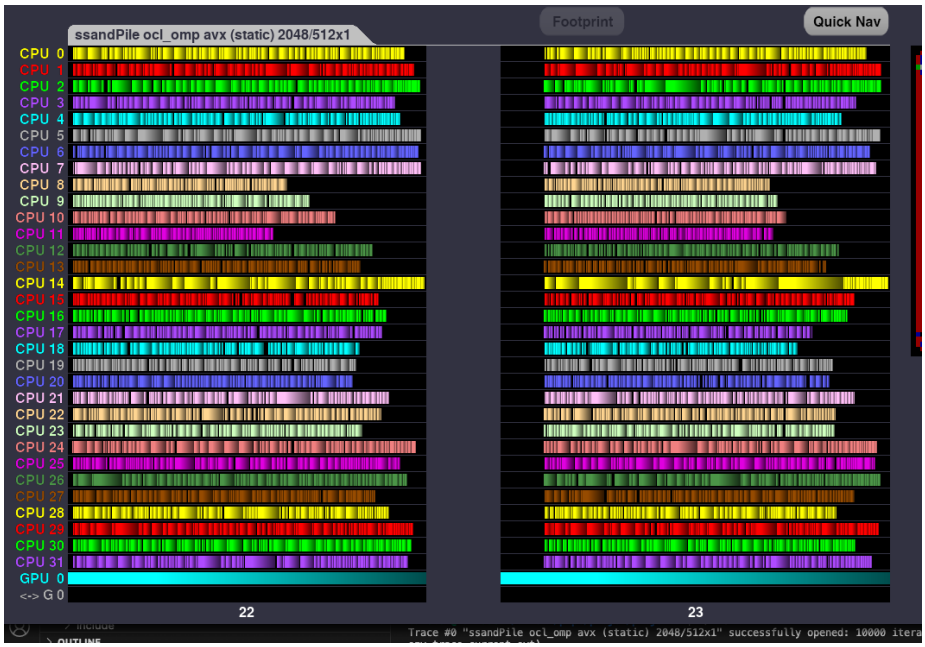

Figure 4: trace 2048

Figure 5: trace 4096

Les premieres iterations sont coûteuses et le cpu commence à travailler tard.