pxpx

Rapport sur le code OpenCL

Programmation des Architectures Parallèles

FURELAUD Thomas

20 Mai 2020

Résumé

Ce rapport présente le code OpenCL établi pour résoudre le problème de l'éboulement d'un tas de sable abélien, ainsi que ses résultats.

Table des matières

1 Version basique

1.1 Code *sable.c*

1.1.1 Fonction *refresh_img*

```
void sable_refresh_img_ocl ()
{
  cl_int err;
  err =
      clEnqueueReadBuffer (queue, cur_buffer, CL_TRUE, 0, sizeof (
          unsigned) * DIM * DIM, TABLE_INT, 0, NULL, NULL);
  check (err, "Failed to read buffer from GPU");
  sable_refresh_img ();
}
```

Cette fonction sert à transférer les valeurs contenues dans cur_buffer directement dans $TABLE\_INT$. Il suffit donc d'un appel à la fonction $clEnqueueReadBuffer$ pour ce faire.

1.1.2 Fonction *sable_invoke_ocl*

```
volatile int index_it = 0;
unsigned sable_invoke_ocl (unsigned nb_iter)
{
    size_t global[2] = {SIZE,SIZE};
    size_t local[2] = {TILEX, TILEY};
    cl_int err;
    cl_mem diff;
    int ret[1];
    ret[0] = 0;

    if(index_it == 0){
        diff = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int),
            NULL, NULL);
        err = clEnqueueWriteBuffer (queue, cur_buffer, CL_TRUE, 0,
            sizeof (int) * DIM * DIM, TABLE_INT, 0, NULL, NULL);
    }

    err = clEnqueueWriteBuffer (queue, diff, CL_TRUE, 0, sizeof(int),
        ret, 0, NULL, NULL);
    check (err, "Failed to write to extra buffer");

    for (unsigned it = 1; it <= nb_iter; it++) {
        err = 0;
        err |= clSetKernelArg (compute_kernel, 0, sizeof (cl_mem), &
            cur_buffer);
        err |= clSetKernelArg (compute_kernel, 1, sizeof (cl_mem), &
            next_buffer);
        err |= clSetKernelArg (compute_kernel, 2, sizeof (cl_mem), &diff
            );
        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");
        if(index_it%10==0){
            err = clEnqueueReadBuffer( queue, diff, CL_TRUE, 0, sizeof(
                unsigned), ret, NULL, NULL, NULL );
            check (err, "Failed to read the mask");
        }
        {
            if(index_it%10==0)
                if(ret[0] == 0){
                  return it;
                }
            cl_mem tmp  = cur_buffer;
            cur_buffer = next_buffer;
            next_buffer = tmp;
        }

    }
    index_it ++;
    return 0;
}
```

Tout d'abord j'ai décidé ici de déclarer une variable volatile $index_it$ qui servira pour savoir à quelle nombre d'itérations nous en somme dans le programme. Celle-ci est alors incrémentée à chaque fin de fonction, juste avant le return.
Lors de la première itération, un nouveau buffer $diff$ est créé. Celui-ci contiendra seulement un int. De plus, lors de celle-ci, l'entièreté de $TABLE\_INT$ est écrit dans le cur_buffer. $TABLE\_INT$ étant simplement le tableau $TABLE$ contenant des int et non des long. En effet, lors du premier appel cur_buffer sera vide, il faut donc le remplir. Cette écriture est nécessaire seulement lors du premier appel, nous verrons par la suite pourquoi.
Ensuite, la valeur du buffer $diff$ est passée à 0. La valeur de ce buffer nous servira pour connaître la stabilité du programme ou non. Ainsi, si en sortie de fonction OpenCL, cette valeur est à 0, cela veut dire qu'il n'y a pas eu de changement et donc que le programme est stable. Nous pourrons alors stopper le programme.
Les 3 arguments de la fonction OpenCL sont alors le cur_buffer pour pouvoir lire $TABLE_INT$, le $next\_buffer$ pour pourvoir récupérer les données, et le buffer $diff$ pour connaître la stabilité du programme.
Pour savoir si le programme est stable, nous lisons alors la valeur de $diff$ et la stockons dans un tableau contenant un élément. Nous vérifions ensuite si cette valeur est égale à 0, si c'est le cas, on retourne alors le nombre d'ité-

rations. Sinon on échange les 2 buffers, de manière à ce que cur_buffer contienne les valeurs actualisées. Pour finir, ce processus est fait seulement toutes les 10 itérations pour éviter le retour sur la RAM de trop d'information. En effet, cette opération (récupération d'un buffer sur la RAM depuis le GPU) étant très cher, du temps est gagné à ne tester la stabilité du programme seulement toutes les 1000 itérations.

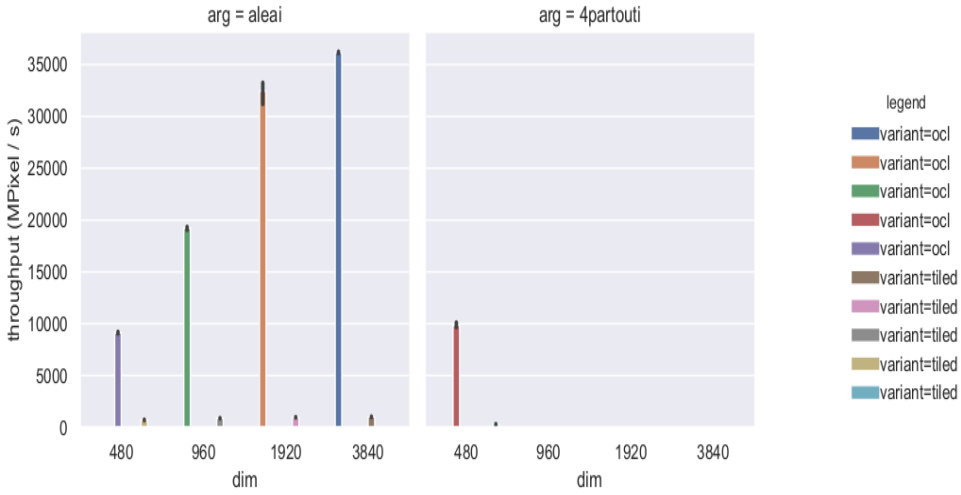
1.2 Code *sable.cl*

1.2.1 Fonction *sabel_ocl*

```
__kernel void sable_ocl (__global unsigned *in, __global unsigned *out,
    __global unsigned *diff)
{

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

  int s = in[y*DIM+x] & 3;

  s+= (y>0) ? in[(y-1)*DIM+x]>>2 : 0;
  s+= (y< DIM-1) ? in[(y+1)*DIM+x]>>2 : 0;
  s+= (x>0) ? in[y*DIM+x-1]>>2 : 0;
  s+= (x < DIM-1) ? in[y*DIM+x+1]>>2 : 0;

  out[y*DIM+x] = s;

  if(in[y*DIM+x] != out[y*DIM+x]){
    diff[0] +=1;
  }

}
```

Cette fonction est assez basique, le modulo 4 de la valeur d'entrée de la case traitée est relevé. Ensuite, on ajoute le résultat de la division par 4 à ce nombre pour chacun de ses 4 voisins, pour lui donner en valeur de retour.
Pour finir, on compare la valeur d'entrée et la valeur de sortie de la case traitée, pour ajouter un à $diff$, si celle ci a changé. Ceci nous servira donc dans le .c. Ici, une barrière n'est pas d'une grande utilité, car la valeur de diff importe peu, nous voulons juste savoir si elle est égale à 0 ou à plus. Ainsi, si toutes les cases sont inchangées, alors diff ne sera pas changée et donc une barrière ne servirait à rien.

1.3 Résultats



pxpx

FIGURE 1 – Comparaison de débit entre la version séquentielle tuilée et la version OpenCL sur les problème 4partout et alea sur différentes dimensions.

Comme on peut le voir sur ce graphique, le gain de temps et de débit est assez conséquent. La version de référence est la version séquentielle tuilée de base. Logiquement, plus la dimension est importante plus le gain de débit l'est aussi. En effet, plus la dimension est grande, plus le nombre de tuile gérée par OpenCL sera important, alors plus le nombre de pixels traités simultanément sera grand.

2 Évaluation paresseuse

2.1 Code de la fonction *sable_invoke_ocl*

Dans cette partie, le nouveau code de *sable_invoke_ocl* est assez similaire à celui de la première partie, mis à part l'ajout de deux nouveaux buffers. Ces buffers serviront à connaître la stabilité d'une tuile pour pouvoir savoir si elle doit être traitée ou pas. Cette technique est assez similaire à celle qu'on

a effectuée dans la première partie du projet, sauf qu'ici l'utilisation de deux buffers est nécessaire pour éviter les concurrences. Dans le premier buffer, initialisé avec que des 0 dedans, signifiant la non stabilité des tuiles, sera stocké la stabilité des cases au début de l'itération. Ainsi c'est celui que nous lirons pour savoir si une tuile doit être traitée ou nous.
Dans le second, initialisée avec des 1 dedans, signifiant donc la stabilité de toutes les tuiles, nous servira à sauvegarder les stabilités des tuiles à la fin de l'itération. Ainsi si une tuile n'est pas stable alors, son index dans ce buffer passe à 1, ainsi que ceux de ses voisins, car si une tuile n'est pas stable alors ses voisins ont des chances de ne pas l'être non plus.
Pour finir, lors du déroulement d'une itération, toutes les cases ne contenant pas de 1 dans le premier buffer sont passées à 1. en effet, nous voulons que le deuxième buffer n'ait que des 1 lors du début de l'itération. Nous remplissons donc le premier buffer, avec des 1 pour ensuite procéder à un swap. Ainsi, le deuxième buffer contiendra que des uns, lors de la prochaine itération, et le premier contiendra la stabilité des tuiles.
Ceci nous donne la nouvelle fonction dans le .cl.

2.2 Code de la fonction *sable_ocl*

```
__kernel void sable_ocl (__global unsigned *in, __global unsigned *out,
    __global int *diff, __global unsigned *read, __global unsigned *write
    )
{
  __local int change;
  change = 0;
  int x = get_global_id(0);
  int y = get_global_id(1);
  int localx = x/(DIM/TILEX);
  int localy = y/(DIM/TILEY);
  if(read[localy*(DIM/TILEY) + localx] != 1){
    int s = in[y*DIM+x] & 3;
    s += (y>0) ? in[(y-1)*DIM+x] >> 2 :0;
    s += (y<DIM-1) ? in[(y+1)*DIM+x] >> 2 :0;
    s += (x<DIM-1) ? in[y*DIM+x+1] >> 2 :0;
    s += (x>0) ? in[y*DIM+x-1] >> 2 :0;
    if(in[y*DIM+x] != s){
      diff[0] +=1;
      change+=1;
    }
    barrier (CLK_LOCAL_MEM_FENCE);
    if(change!=0){
      write[localy*(DIM/TILEY) + localx] = 0;
      write[(localy-1)*(DIM/TILEY) + localx] = 0;
      write[(localy+1)*(DIM/TILEY) + localx] = 0;
      write[localy*(DIM/TILEY) + localx-1] = 0;
      write[localy*(DIM/TILEY) + localx+1] = 0;
```

```
    }
    read[localy*(DIM/TILEY) + localx] = 1;
    out[y*DIM+x] = s;
  }else{
    out[y*DIM+x] = in[y*DIM+x];
  }
}
```

2.3 Résultats

Les résultats de cette méthode ne sont pas vraiment bons. En effet premièrement, l'image en sortie n'est pas la bonne, puis les temps pour résoudre les problèmes, sont aussi lent, voire beaucoup plus lent pour les grandes dimensions. On peut donc en conclure, qu'une ou plusieurs erreurs de calculs sont présentes ici, ainsi qu'une opération trop coûteuse.