Université de Bordeaux

ind-47:
Rendu OpenMP/OpenCL

MENADJLIA SOFIANE

May 3, 2022

Contents

1 Version Hybride OpenMP/OpenCL

- La version hybride OpenMP/OpenCL consiste a diviser la charge du travail du noyau synchrone entre le CPU ainsi que le GPU afin de maximiser le rendement.

- Voici donc les différentes parties de l'implémentation:

1.1 Initialisation du contexte

- Afin d'implémenter cette version hybride il suffit de configurer la queue d'exécution du GPU au sein de la fonction **ssandPile_init_ocl_hybrid**.

- Après avoir appelé de la fonction **ssandPile_init**, la répartition de charges doit aussi être faite dans cette fonction.

1.1.1 Code

- cpu_y_part et gpu_y_part représentent les parties a faire par le CPU et le GPU respectivement.

- size représente le nombre de tuiles dans la partie a faire par le GPU.

- TABLE_BUFF et new_buff sont initialisé afin de permettre la détection de la terminaison du programme.

```
1
2  void ssandPile_init_ocl_hybrid (void)
3  {
4    ssandPile_init();
5
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
10   cpu_y_part = (NB_TILES_Y / 2) * GPU_TILE_H; // Start with fifty-fifty
11   gpu_y_part = DIM - cpu_y_part;
12
13   int size = (DIM / GPU_TILE_W) * (gpu_y_part / GPU_TILE_H);
14   TABLE_BUFF = (TYPE *) malloc(size * sizeof(TYPE));
15
16   if (!TABLE_BUFF)
17     exit_with_error ("unable to allocate new buffer table!!\n");
18
19   new_buff = clCreateBuffer (context, CL_MEM_READ_WRITE,
20                              sizeof (TYPE) * size , NULL, NULL); // to check size
21
22   if (!new_buff)
23     exit_with_error ("unable to allocate new buffer!!\n");
24 }
```

1.2 Détection de terminaison

- La version OpenCL du rendu précèdent n'était pas fini car elle n'arrivait pas a détecter la stabilisation du tas de sable, il fallait donc préciser le bon nombre d'itérations a effectuer au lancement du programme.

- Étant donne que la version hybride est exécutée sur deux supports diffèrent il faut qu'il communiquent l'information sur l'état de stagnation courant a travers la variable change.

- Il se trouve que en OpenCL la lecture et écriture dans un buffer est très coûteuse, il faut donc trouver un système pour éviter de récupérer cette information a chaque itération.

- Dans ma version j'ai choisi de vérifier cela chaque **DIM/8** car ça donne le meilleur rendement avec différentes tailles d'images.

1.2.1 code

- Je commence par initialiser TABLE_BUFF et new_buff dans la fonction **ssandPile_init_ocl_hybrid** avec le même **size = (DIM / GPU_TILE_W) * (gpu_y_part / GPU_TILE_H)** qui représente le nombre de tuiles dans la partie GPU.

- Chaque élément dans ce buffer contient le changement cumulé de toute les cellules dans la tuile correspondante.

- Après avoir exécuté le code des deux parties on test que la variable change est a 0 dans la partie CPU ainsi que le nombre d'itérations est bien divisible par (DIM/8) avant de lire les changements du cote GPU en faisant un READ de new_buff dans TABLE_BUFF.

- Enfin il suffit de parcourir TABLE_BUFF et récupérer le changement total dans la variable change qu'on pourra après tester pour savoir si le programme s'est terminé.

```
1   if (change == 0 && iteration % (DIM/8) == 0){
2     err = clEnqueueReadBuffer (queue, new_buff, CL_TRUE, 0, sizeof(unsigned
        ) * size, TABLE_BUFF, 0, NULL, NULL);
3
4     check (err, "Failed to read to buffer");
5
6     #pragma omp parallel for schedule(runtime) reduction(|: change)
7     for (int i = 0; i < size ; i++){
8       change |= TABLE_BUFF[i];
9     }
10
11    if (!change){
12      diff = it;
13      break;
14    }
15  }
```

1.3 Appel du traitement GPU

- Cette partie concerne l'implémentation du code du cote CPU qui permet d'invoquer le kernel OCL du cote GPU.

- Pour cela, après avoir reparti la charge de travail, il faut ajouter les arguments a passer au kernel OCL puis l'appeler.

1.3.1 Code

Les paramètres a passer sont les suivants:

- **cur_buffer** : buffer en cours d'utilisation
- **next_buffer** : buffer de la prochaine itération
- **new_buffer** : buffer permettant de détecter la terminaison du programme
- **cpu_y_part** : servant de offset pour le kernel afin de ne pas commencer au début de limage.
- **itération** : permettant de vérifier les itérations avant décrire dans **new_buff**

```
    // Set kernel arguments

    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), &new_buff);
    err |= clSetKernelArg (compute_kernel, 3, sizeof (unsigned), &cpu_y_part);
    err |= clSetKernelArg (compute_kernel, 4, sizeof (unsigned), &iteration);

    check (err, "Failed to set kernel arguments");

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

    clFlush (queue);
```

1.4 Traitement CPU

- Le traitement du cote CPU est fait grâce a OpenMP et est très similaire a ceux des rendus précédents, avec l'exception de parcourir les lignes jusqu'à **cpu_y_part** au lieu de **DIM** pour ce limiter a la partie CPU.

- les variable t1 et t2 permettent de connaître la durée de temps de calcul de la partie cpu.

1.4.1 Code

```
    t1 = what_time_is_it ();
    int change = 0;
    #pragma omp parallel for collapse(2) schedule(runtime) reduction (|: change)
    for (int y = 0; y < cpu_y_part; y += TILE_H)
      for (int x = 0; x < DIM; x += TILE_W)
        change |=
            do_tile(x + (x == 0), y + (y == 0),
                    TILE_W - ((x + TILE_W == DIM) + (x == 0)),
                    TILE_H - ((y + TILE_H == DIM) + (y == 0)), omp_get_thread_num
                       ());

    t2 = what_time_is_it ();
```

1.5 Communication et échange de données

- Afin de permettre le bon fonctionnement de programme il faut que le kernel OCL communique les résultat des données traitées au traitement CPU et vice versa.

- Pour cela il suffit de communiquer les deux lignes correspondantes a l'intersection entre les deux parties, notamment les lignes **cpu_y_part** et **gpu_y_part**.

- Concrètement, j'utilise les fonctions **clEnqueueReadBuffer** et **clEnqueueWriteBuffer** qui servent a lire la première ligne du renvoyée par le gpu ainsi qu'écrire la dernière ligne traitée par le cpu.

1.5.1 Code

```
1   //READ WRITE
2
3   err = clEnqueueReadBuffer (queue, next_buffer, CL_TRUE, (cpu_y_part) * DIM *
        sizeof(unsigned) ,
4                              DIM * sizeof (unsigned), &table(out, (cpu_y_part),
                                   0), 0,
5                              NULL, NULL);
6
7   check (err, "Failed to read to buffer");
8
9   err = clEnqueueWriteBuffer (queue, next_buffer, CL_TRUE, (cpu_y_part - 1) *
        DIM * sizeof(unsigned) ,
10                             DIM * sizeof (unsigned), &table(out, (cpu_y_part -
                                   1), 0), 0,
11                             NULL, NULL);
12
13  check (err, "Failed to write to buffer");
```

1.6 Kernel OpenCL GPU

- Le traitement du cote GPU est fait grâce a OpenCL et est très similaire a celui du rendu OCL précèdent.

- Comme pour la partie CPU, il faut commencer a partir de la bonne ligne par rapport a la partie désigné pour le traitement GPU.

- il faut aussi gérer l'écriture dans le buffer **new_buff** pour permettre la lecture du cote CPU.

1.6.1 Code

- Le code ne diffère pas trop sauf qu'il faut incrémenter y de **offset** qui vaut toujours **cpu_y_part**, avant de commencer afin de se limiter a la partie GPU.

- Il faut trouver la tuile correspondante de chaque cellule en normalisant le x et y avant de pouvoir mettre le cumul du changement total dans le buffer **new_buff**.

```
1
2  __kernel void ssandPile_ocl_hybrid (__global unsigned *in, __global unsigned *out
       , __global unsigned *new, unsigned offset, unsigned iteration)
3  {
4    int y = get_global_id (1) + offset;
5    int x = get_global_id (0);
6
7    int index = y * DIM + x;
8
9    unsigned local_y = get_local_id (1);
```

```
  unsigned local_x = get_local_id (0);

  if(x > 0 && y > 0 && x < DIM-1 && y < DIM-1)
  {

    out[index] =
    (in[y * DIM + x] % 4) +
    (in[(y+1) * DIM + x] /4) +
    (in[(y-1) * DIM + x] /4) +
    (in[y * DIM + (x+1)] /4) +
    (in[y * DIM + (x-1)] /4);

  barrier (CLK_LOCAL_MEM_FENCE);

  y = y / GPU_TILE_H;
  x = x / GPU_TILE_W;
  int tile_index = y * (DIM / GPU_TILE_W) + x;
  if (iteration % (DIM/8) == 0)
    new[tile_index] |= in[index] != out[index];
  }
}
```

2 Performances

2.1 Carte de chaleur

- En ce qui concerne cette version hybride on remarque un speedup assez grand avec un threading de 44 a 48.
- on remarque que la version statique donne un rendement nettement supérieure a celui de la dynamique.
- Le tuilage optimale et de 32 * 16 avec 44 threads, 16 * 32 avec 46 et 32 * 16 avec 48.

Figure 1: ssandPile version ocl_hybrid avec do_tile_opt size 2048 mode alea tile size 32*16 machine dali(48 threads)

2.2 Trace

Voici la trace généré grâce a l'option -t de EasyPAP ou on peut voir que les threads cpu ne sont très bien exploités et prennent beaucoup de temps par rapport au gpu.

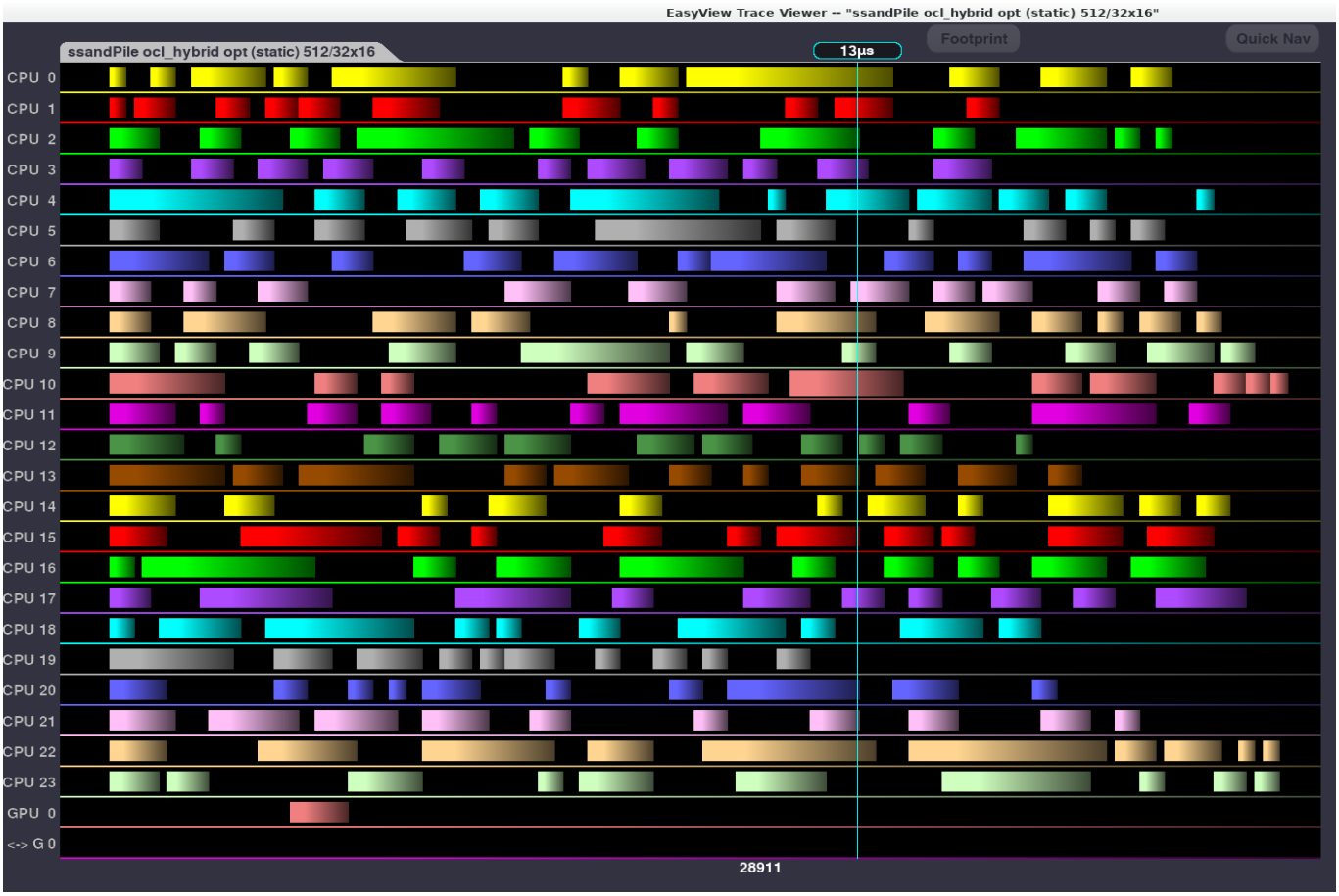

Figure 2: trace-ssand-omp-tiled-512/32x16

3 Rapport de bugs

- Malheureusement, je n'ai pas réussi a implémenter la partie concernant le rééquilibrage de charges, même après avoir changé les valeurs **cpu_y_part** et **gpu_y_part** ainsi que mis a jour le tableau global et avoir ré-alloué les buffers de terminaison avec la bonne taille.

- le programme dans ce cas semble de seulement marcher correctement quand appelé avec l'option "-i" et ceci est du au fait que le programme est appelé plusieurs fois avec un nb_iter mis a 1 ce qui impose l'appel de la fonction **ssandPile_init_ocl_hybrid** avec la répartition de travail mise a 50% pour chacun.

- Même en initialisant la valeur **cpu_y_part** a **(NB_TILES_Y / 4) * GPU_TILE_H** au lieu de **(NB_TILES_Y / 2) * GPU_TILE_H**.

- Le code correspondant au bugs mentionnés si-dessus est commenté sur le code.

3.1 Code

```
1
2   // Load balancing
3   // if (gpu_duration != 0) {
4     // if (much_greater_than (gpu_duration, cpu_duration) &&
5     // gpu_y_part > GPU_TILE_H) {
6     // gpu_y_part -= GPU_TILE_H;
7     // cpu_y_part += GPU_TILE_H;
8     // global[1] = gpu_y_part;
9     // }
10    // else
11  // if (much_greater_than (cpu_duration, gpu_duration) &&
12  // cpu_y_part > GPU_TILE_H) {
13  // gpu_y_part += GPU_TILE_H;
14  // cpu_y_part -= GPU_TILE_H;
15  // global[1] = gpu_y_part;
16  // }
17  // }
18  size_t size = (DIM / GPU_TILE_W) * (gpu_y_part / GPU_TILE_H);
19  // TABLE_BUFF = realloc(TABLE_BUFF, size * sizeof(TYPE));
20  // cl_mem new = clCreateBuffer (context, CL_MEM_READ_WRITE, sizeof (TYPE) *
        size , NULL, NULL);
```

4 Conclusion

- Dans ce rendu j'ai réussi à implémenter la version hybride OpenMP et OpenCL indiqué dans le rendu 4 ainsi que la détection de terminaison du programme.

- Je tiens a préciser que j'ai appris beaucoup sur la programmation parallèle grâce a cette UE, ça ma permis de mieux connaître mes préférences ainsi qu'approfondir mes compétences dans le domaine.