Programmation Parallèle

Fiche 7 : Premiers pas en OpenCL

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/pap/

Il s'agit de s'initier au calcul vectoriel en utilisant le langage OpenCL qui permet de programmer les accélérateurs de calcul tels que les cartes graphiques (GPU). Comme à l'accoutumée, récupérez les dernières mises à jour à l'aide d'un discret `git pull`.

1 Préambule

OpenCL est à la fois une bibliothèque et une extension du langage C permettant d'écrire des programmes s'exécutant sur une (ou plusieurs) cartes graphiques. Le langage OpenCL est très proche du C, et introduit un certain nombre de qualificateurs parmi lesquels :

- `__kernel` permet de déclarer une fonction exécutée sur la carte et dont l'exécution peut être sollicitée depuis les processeurs hôtes
- `__global` pour qualifier des pointeurs vers la mémoire globale de la carte graphique
- `__local` pour qualifier une variable partagée par tous les threads d'un même *workgroup*

La carte graphique ne peut pas accéder [1] à la mémoire du processeur, il faut donc transférer les données dans la mémoire de la carte avant de commencer un travail. La manipulation (allocation, libération, etc.) de la mémoire de la carte se fait par des fonctions spéciales exécutées depuis l'hôte :

- `clCreateBuffer` pour allouer un tampon de données dans la mémoire de la carte ;
- `clReleaseMemObject` pour le libérer ;
- `clEnqueueWriteBuffer` et `clEnqueueReadBuffer` pour transférer des données respectivement depuis la mémoire centrale vers la mémoire du GPU et dans l'autre sens.

Vous trouverez une synthèse utile des primitives OpenCL 1.2 ici :

https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

2 Vérification de la configuration matérielle de votre machine

Placez-vous dans votre répertoire de travail EASYPAP, et demandez l'affichage d'informations OpenCL comme ceci : `./run --show-ocl`. Vous devriez obtenir la liste des périphériques supportés par OpenCL :

```
3 OpenCL platforms detected
Platform 0: NVIDIA CUDA (NVIDIA Corporation)
+++ Device 0 : GPU [GeForce RTX 2070]
Platform 1: Portable Computing Language (The pocl project)
```

1. En tout cas, pas de manière efficace au travers du bus PCI...

```
--- Device 0 : CPU [pthread-Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz]
Platform 2: Intel(R) OpenCL (Intel(R) Corporation)
--- Device 0 : CPU [Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz]
Note: OpenGL renderer uses [GeForce RTX 2070]
```

Vérifiez que la cible choisie (ligne débutant par `+++`) soit bien la carte graphique utilisée par OpenGL (dernière ligne). Si ce n'est pas le cas, utilisez les variables d'environnement `PLATFORM` et/ou `DEVICE` pour choisir le dispositif cible.

Par exemple, si vous désirez utiliser les processeurs Intel Xeon avec le compilateur Intel [2], Vous préfixerez l'exécution d'EASYPAP de la façon suivante :

```
PLATFORM=2 ./run --show-ocl
```

Ou positionnez la variable une bonne fois pour toute la durée de la session au moyen d'un redoutable : `export PLATFORM=2`

3 Premier noyau

Notre premier noyau OpenCL s'appelle `invert` et il calcule le « négatif » d'une image en calculant le complément à 1 des composantes (R,V,B) de chaque pixel. Son implémentation se trouve dans `kernel/ocl/invert.cl`.

Essayez-le avec une pause entre chaque itération :

```
./run -l images/shibuya.png -k invert -o -p
```

Inspirez-vous de cette implémentation pour créer `sample.cl`, la variante OpenCL du noyau `sample` qui colorie simplement uniformément l'image courante. Attention, au contraire du noyau `invert`, le noyau `sample` n'utilise qu'une seule image, donc un seul paramètre est suffisant. Observez comment ce noyau est lancé depuis le CPU en examinant la fonction `sample_invoke_ocl` (dans `kernel/c/sample.c`).

Pour tester votre noyau, tapez simplement :

```
./run -k sample -o
```

Comme `invert`, votre noyau fait l'hypothèse qu'il est exécuté avec un thread par pixel de l'image. Observez ce qui se passe si vous lancez :

```
SIZE=512 ./run -k sample -o
```

Vous pouvez changer le nombre de threads lancés selon une dimension en modifiant la fonction `sample_invoke_ocl`. Par exemple :

```
unsigned sample_invoke_ocl (unsigned nb_iter)
{
  size_t global[2] = {GPU_SIZE_X / 2, GPU_SIZE_Y};
  size_t local[2]  = {GPU_SIZE_X, GPU_SIZE_Y};
  ...
```

2. Mais vous ne pourrez pas obtenir d'affichage interactif. Vous pourrez juste tester les performances avec `-n`...

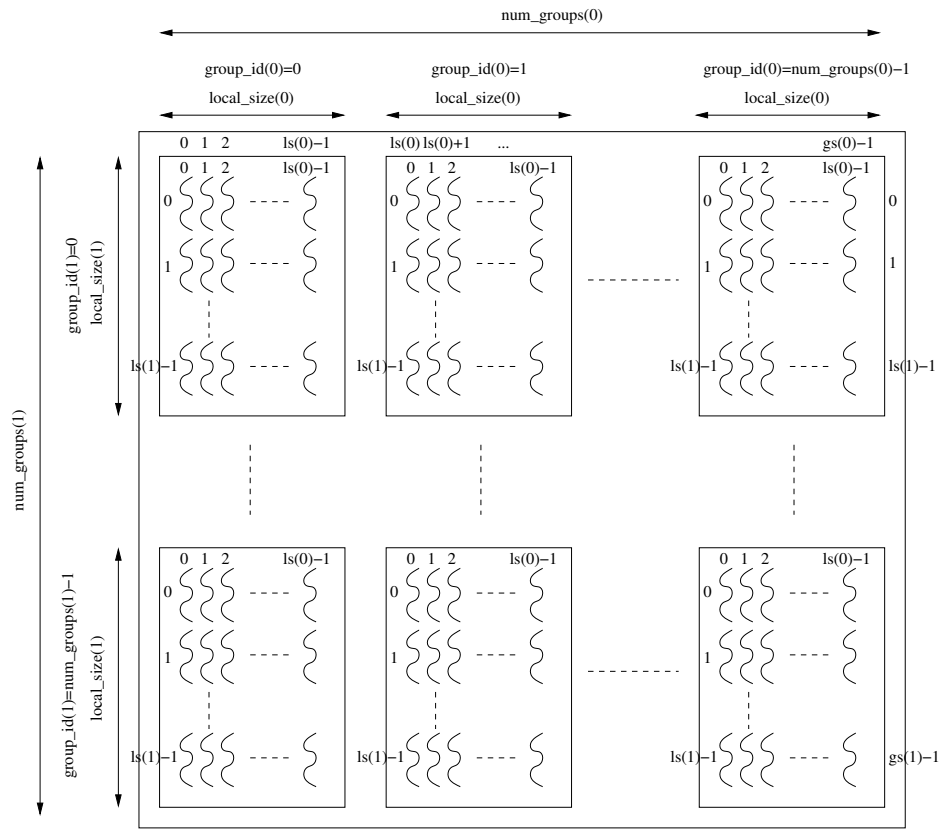

FIGURE 1 – Threads et workgroups

4 Notion de workgroup

Lors du lancement d'un noyau OpenCL, il faut indiquer combien de threads on veut créer selon chaque dimension (la numérotation des threads peut s'effectuer selon 1, 2 ou 3 dimensions). Il faut également spécifier la façon dont vous voulez repartir vos threads, c'est-à-dire spécifiez la taille (selon chaque dimension) des *workgroups*. Tous les threads d'un même workgroup (par exemple de $16 \times 16 = 256$ threads) ont l'assurance d'être placés sur le même *Shared Multiprocessor*. Plus tard, nous verrons qu'ils pourront partager de la mémoire locale entre eux et se synchroniser.

À l'intérieur d'un noyau exécuté par le GPU, des variables sont prédéfinies afin de connaître les coordonnées absolues ou relatives au *workgroup* dans lequel le thread se trouve, ou encore les dimensions des *workgroups* :

get_num_groups (d) : dimension de la grille de workgroups (selon la d^{ieme} dimension)

get_group_id (d) : position du workgroup courant

get_global_id (d) : position absolue du thread courant

get_global_size (d) : nombre labsolu de threads

get_local_id (d) : position relative du thread à l'intérieur du workgroup courant

get_local_size (d) : nombre de thread par workgroup

Cette numérotation est illustrée en figure 1 (page 3).

4.1 Observation de la forme des workgroups

Ajoutez la ligne suivante dans votre noyau `sample_ocl` :

```
...
if ((get_group_id (0) + get_group_id (1)) % 2)
  img [y * DIM + x] = ...;
...
```

Observez le résultat à l'écran. Utilisez des workgroups rectangulaires, comme par exemple :

```
TILEY=8 TILEX=32 ./run -k sample -o
```

Essayez plusieurs paramètres. Quelle est la taille maximale d'un workgroup sur le GPU ?

4.2 Influence des workgroups sur la performance

En utilisant le noyau `invert`, examinez l'influence sur les performances de la forme des workgroups. En conservant une taille de workgroup équivalente, comparez par exemple :

```
TILEY=16 TILEX=32 ./run -l images/shibuya.png -k invert -o -i 1000 -n
```

```
TILEY=1 TILEX=512 ./run -l images/shibuya.png -k invert -o -i 1000 -n
```

```
TILEY=512 TILEX=1 ./run -l images/shibuya.png -k invert -o -i 1000 -n
```

Intrigant non ?

5 Divergence

L'objectif est de mesurer l'influence d'un saut conditionnel sur les performances d'un code GPU. Pour cela, nous allons utiliser un noyau pour lequel la moitié des threads ne fait pas le même calcul que l'autre...

Ce noyau, c'est `stripes`, un noyau qui éclaircit/obscurcit certaines parties d'une image en fonction du numéro de colonne. Commencez par examiner me code OpenCL du noyau (fichier `kernel/c/stripes.cl`) et observer comment est calculée la variable `mask`.

Le paramètre `PARAM` correspond à l'argument passé en ligne de commande après l'option `-a`. Voici comment tester le programme avec `PARAM=4` :

```
TILEY=1 TILEX=128 ./run -l images/1024.png -k stripes -o -a 4
```

Observez visuellement la différence entre les résultats obtenus pour les valeurs $1, 2, 3, 4, ...8$ pour `PARAM`. Pour une valeur n, combien de threads « consécutifs » effectuent le même traitement ?

Obvervez l'impact sur les performances des différentes valeurs de `PARAM` (de 1 à 8) :

```
TILEY=1 TILEX=128 ./run -l images/1024.png -k stripes -o -i 1000 -n -a 8
```

Déterminez la valeur charnière. Conclusion ?

En utilisant des workgroups carrés (16×16 par défaut), le comportement conforte-t-il votre hypothèse ?

```
./run -l images/1024.png -k stripes -o -i 1000 -n -a 4
```

5.1 Utilisation de la mémoire locale

L'objectif est de calculer la rotation d'une image à 90° vers la gauche.

La version qui vous est fournie (dans `rotation90.cl`) est une version « naive » manipulant directement la mémoire globale. Elle se lance simplement :

```
./run -l images/shibuya.png -k rotation90 -o
```

1. Expliquez pourquoi cette version ne peut pas être très performante (appuyez vous sur une situation analogue étudiée en cours).
2. En utilisant un tampon de taille[3] `GPU_TILE_H` × `GPU_TILE_W` en mémoire locale au sein de chaque *workgroup*, arrangez-vous pour que les lectures *et* les écritures mémoire soient correctement *coalescées*.
3. Est-il utile d'utiliser une barrière `barrier(CLK_LOCAL_MEM_FENCE)` pour synchroniser les threads d'un même workgroup ?
4. Que se passe-t-il si on utilise un tableau temporaire de dimensions `[GPU_TILE_H]`×`[GPU_TILE_W + 1]` ?

5.2 Pixellization

On souhaite à présent appliquer un effet de pixellization aux images, en s'arrangeant pour que les images soient constitués de blocs carrés de taille `PIX_BLOC`x`PIX_BLOC` de couleur uniforme.

Pour simplifier, on supposera que `PIX_BLOC` est inférieur ou égal à 32.

5.2.1 Echantillonnage

Lancez le noyau `pixelize` qui vous est fourni comme base de départ :

```
./run -l images/shibuya.png -k pixelize -o
```

Ok, ça pixellise... Mais comment ? Inspectez le code de `pixelize_ocl` pour comprendre d'où vient la couleur de chaque bloc.

5.2.2 Meilleure pixellisation

Pour obtenir un résultat plus fidèle à l'image d'origine, il faut calculer la *couleur moyenne* de tous les pixels d'un même bloc, puis affecter cette moyenne à tous les pixels du bloc.

Autrement dit, il faut faire une réduction... À vous de jouer !

3. les constantes `GPU_TILE_H` et `GPU_TILE_W` sont positionnées à 16 par défaut, mais peuvent être modifiées via les variables d'environnement du shell `TILEY` et `TILEY`. En outre, elle sont transmises au noyau OpenCL sous forme de constantes lors de la compilation.