Calcul vectoriel sur GPU : OpenCL

Il s'agit de s'initier au calcul vectoriel grâce à OpenCL qui permet de programmer les cartes graphiques des ordinateurs du CREMI. Vous trouverez des ressources utiles dans le répertoire :

`/net/cremi/rnamyst/etudiants/pmg/TP6-OpenCL/`

Dans la plupart des premiers programmes d'exemples qui vous sont fournis, les options en ligne de commande suivantes sont disponibles :

`prog { options } <tile1> [<tile2>]`
`options:`

`-g | --gpu-only`	Exécute le noyau OpenCL uniquement sur GPU même si une implémentation OpenCL est disponible pour les CPU
`-s <n> | --size <n>`	Exécute le noyau OpenCL avec n threads ($n \times n$ si le problème est en 2D). Il est possible de spécifier des kilo-octets (avec le suffixe k) ou des méga-octets (avec le suffixe m). Ainsi, `-s 2k` est équivalent à `-s 2048`.

`tile` permet de fixer la taille du workgroup à tile1 threads (tile1 × tile1 ou bien tile1 × tile2 threads si le problème est en 2D).

1 Découverte

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 des primitives OpenCL utiles dans un « *Quick Reference Guide* » situé ici :

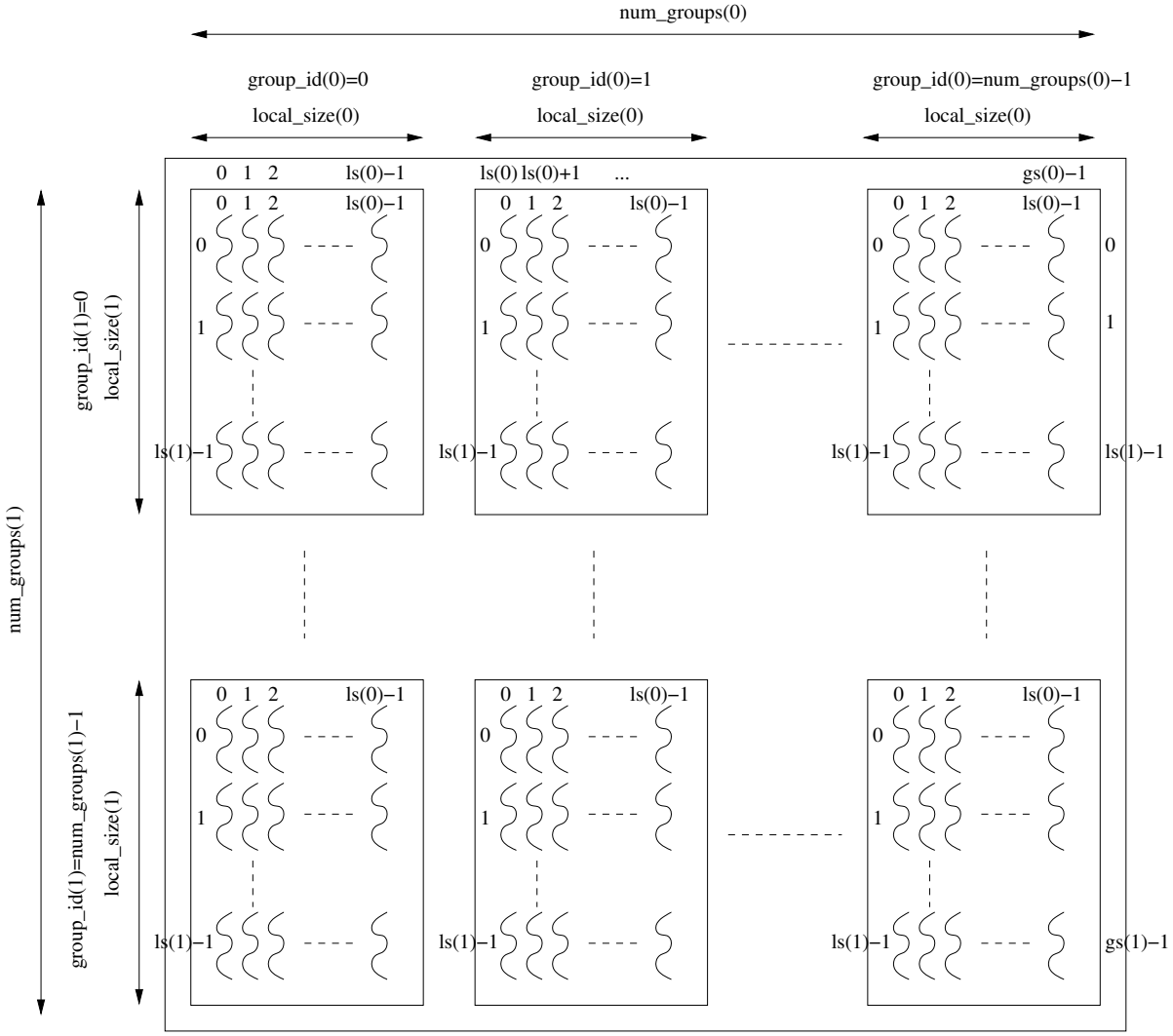
`/net/cremi/rnamyst/etudiants/opencl/Doc/opencl-1.2-quick-reference-card.pdf`

Lorsqu'on execute un « noyau » sur une carte graphique, il faut indiquer combien de threads on veut créer selon chaque dimension (les problèmes peuvent s'exprimer selon 1, 2 ou 3 dimensions), et de quelle manière on souhaite regrouper ces threads au sein de *workgroups*. Les threads d'un même workgroup peuvent partager de la mémoire locale, ce qui n'est pas possible entre threads de workgroups différents. À l'intérieur d'un noyau exécuté par le GPU, des variables sont définies afin de connaître les coordonnées absolues ou relatives au *workgroup* dans lequel le thread se trouve, ou encore les dimensions des *workgroups* :

1. En tout cas, pas de manière efficace

`get_num_groups(d)` : dimension de la grille de workgroups selon la d^{ieme} dimension
`get_group_id(d)` : position du workgroup courant selon la d^{ieme} dimension
`get_global_id(d)` : position absolue du thread courant selon la d^{ieme} dimension
`get_global_size(d)` : nombre labsolu de threads selon la d^{ieme} dimension
`get_local_id(d)` : position relative du thread à l'intérieur du workgroup courant selon la d^{ieme} dimension
`get_local_size(d)` : nombre de thread par workgroup selon la d^{ieme} dimension

Le dessin suivant montre ceci de manière visuelle.



2 Multiplication d'un vecteur par un scalaire

Regardez le code source du noyau dans `vector.cl`. Remarquez qu'on fait travailler les threads adjacents sur des éléments adjacents du tableau : contrairement à ce qu'on a vu pour les CPUs, dans le cas des GPU c'est la meilleure façon de faire, car les threads sont ordonnancés sur un multiprocesseur par paquets de 32 (ces paquets appelés *warp*) : ils lisent ensemble en mémoire (lecture dite *coalescée*) et calculent exactement de la même façon. Jouez avec la taille des *workgroups* (4, 8, 16, 32,...) en sachant qu'un workgroup ne peut pas contenir plus de 1024 éléments sur nos cartes.
Faire en sorte que le kernel `vector.cl` implémente le produit d'un vecteur par un scalaire. Modifiez ensuite ce code pour que chaque thread traite l'élément d'indice `(get_global_id(0)+16)` modulo le nombre de threads. Normalement, le programme doit encore fonctionner.

Le paramètre du `vector TILE` permet de modifier la taille des *workgroups* employés . Exécutez le programme en jouant avec la taille des *workgroups* sans dépasser les 1024 éléments (limite de nos cartes). Quelle taille donne les meilleures performances ?

3 Addition de matrices

Modifier le kernel Le programme `addMat.cl` afin d'effectuer une addition de matrices.Lors d'un appel `addMat TILE1 TILE2` le calcul est structuré en deux dimensions : les workgroups sont constitués de `TILE1` $\times$ `TILE2` threads.
Exécutez le programme en jouant avec la taille des *workgroups* sans dépasser les 1024 éléments (limite de nos cartes). Comparer les performances obtenues pour différentes décompositions de 256 (256×1, 128×2, 64×4,..., 1×256) .

4 Traitements sur des images

L'objectif est maintenant d'utiliser OpenCL pour travailler sur des matrices 2D de pixels (des images quoi). Pour tous les exercices qui suivent, on se placera dans le sous-répertoire `fichiers/Images`. L'unique fichier contenant les noyaux OpenCL est : `kernel/compute.cl`.

4.1 Prise en main

Commencez par inspecter le code du noyau « `scrollup` », dont la finalité est de décaler l'image d'une ligne vers le haut.
Voici comment lancer l'exécution de ce noyau sur la carte graphique :

```
./prog -l images/shibuya.png -k scrollup -v ocl
```

En cas de problème, typiquement si la plateforme utilisée par défaut est `Intel` et non `Nvidia`, positionnez la variable d'environnement `PLATFORM` comme ceci :

```
PLATFORM=1 ./prog -l images/shibuya.png -k scrollup -v ocl
```

Par défaut, le noyau est exécuté par `DIM`^{2} *workitems*. La variable d'environnement `SIZE` vous permet de modifier le nombre de *workitems* lancés (`SIZE`^{2} dans ce cas). Essayez :

```
SIZE=1024 ./prog -l images/shibuya.png -k scrollup -v ocl
```

Notez que les constantes DIM et SIZE sont toutes deux disponibles dans les noyaux OpenCL. ON aurait donc pu utiliser `SIZE` au lieu de `get_global_size (1)` dans le cas présent.

4.2 Notre premier traitement d'image

Nous allons programmer un filtre d'inversion vidéo, qui modifie chaque pixel de l'image en calculant le complémentaire (à 255) de chacune ses composantes Rouge, Vert et Bleu.
Comme vous l'avez remarqué en observant le noyau `scrollup`, une image est une matrice d'entiers non signés. Ces entiers codent un quatruplet de quatre octets : Rouge, Vert, Bleu, Alpha.
Ecrivez un noyau OpenCL « `invert` » (par copier-coller à partir de `scrollup`) qui se contente d'appliquer le traitement suivant à chaque pixel :

```
couleur |= 0xFF000000;
```

Qu'en déduisez-vous ? Essayez avec la constante `0xFF0000`, puis avec `0xFF00`.
Maintenant que vous avez repéré la position des octets codant les composantes Rouge, Vert et Bleu, écrivez une expression inversant ces trois composantes pour réaliser un « négatif ».
Pour mesurer les performances, vous pouvez lancer :

```
./prog -l images/shibuya.png -k invert -v ocl -n -i 500
```

Vous pouvez tester des tailles de workgroups différentes, par exemple :

```
TILEX=32 TILEY=8 ./prog -l images/shibuya.png -k invert -v ocl -n -i 500
```

4.3 Effet de la divergence sur les performances

L'objectif est de mesurer l'influence d'un saut conditionnel sur les performances. Que se passe-t-il lorsque la moitié des threads ne fait pas le même calcul que l'autre ?

1. Modifiez le noyau `strip` de sorte que les pixels de coordonnée x paire soient éclaircis et ceux de coordonnées x impaire soient foncés.
   Mesurez les performances observées par rapport à la version d'origine.
2. Modifiez le code pour éclaircir les pixels pour lesquels `x & mask` est vrai, et obscurcir ceux pour lesquels `x & mask` est faux. Si test vaut 1, le code est équivalent au point précédent. Insérez le code suivant au début du noyau `strip` :

   ```
   #ifdef PARAM
     unsigned mask = PARAM;
   #else
     unsigned mask = 1;
   #endif
   ```

   Vous pouvez maintenant modifier la valeur de `mask` depuis la ligne de commande (ici 4) :

   ```
   ./prog -l images/1024.png -k strip -v ocl -p 4
   ```

   Testez successivement les puissances de deux, de 1 à 64. Constatez la différence à l'écran. Puis mesurez les performances. Que constatez-vous ?

4.4 Transposition de matrice.

L'objectif est de calculer la transposée d'une matrice, c'est-à-dire d'exécuter `out[i][j] = in[j][i]` pour chaque élément de la matrice `in`.
La version qui vous est fournie (noyau `transpose_naif`) est une version manipulant directement la mémoire globale.

1. Expliquez pourquoi cette version ne peut pas être très performante (appuyez vous sur les expériences réalisées sur la somme de matrices).
2. En utilisant un tampon de taille [2] `TILEY`×`TILEX` 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 `TILEY`×`TILEX + 1` ?

4.5 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.

4.5.1 Echantillonnage

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

```
./prog -l images/shibuya.png -k tiles -v ocl
```

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

4.5.2 Meilleure pixellization

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 !

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