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/opencl/`.

Pour les TP et projets nous utiliserons prioritairement les cartes graphiques de Nvidia kepler K2000 des salles 203/008 et la carte K20Xm du serveur tesla. La carte K2000 est équipée de deux multipocesseurs de 192 cœurs chacun, la carte K20Xm dispose quant à elle de quatorze multipocesseurs.

Dans la plupart des (squelettes de) 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 :

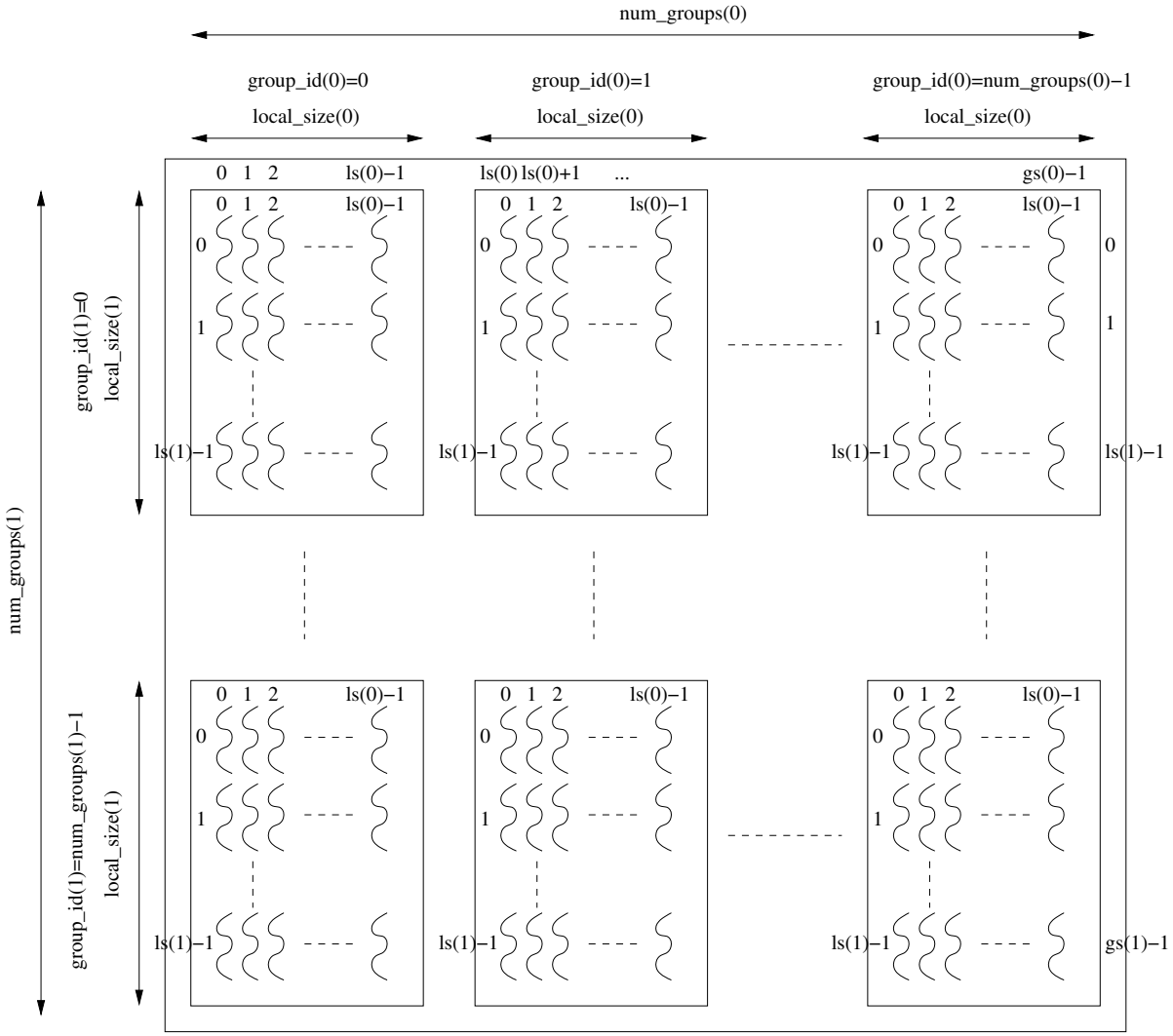
`/net/cremi/rnamyst/etudiants/opencl/Doc/opencl-1.2-quick-reference-card.pdf`

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.

1. En tout cas, pas de manière efficace

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* :
`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` × `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 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 `vector.cl` de sorte que chaque thread exécute 10 fois la multiplication pour bien mettre en valeur le phénomène.
2. Dupliquer le répertoire afin de produire une version où les threads pairs calculent 10 fois une multplication par le paramètre `k`, les autres multipliant par `3.14`.
3. Faire une deuxième version où les groupes pairs calculent les 10 multplications par `k`, les autres celles par `3.14`.
4. Faire une troisième version où les threads dont le numéro a son ième bit à 1 (`(index >>i)&1`) `0` calculent les 10 multplications par `k`, les autres celles par `3.14`. Pour quelle plus petite valeur de `i` obtient on de bonnes performances ?

5 Transposition de matrice.

L'objectif du programme `Transpose` est de calculer la transposée d'une matrice. Il s'agit « simplement » de calculer `B[i][j] = A[j][i]`. La version qui vous est fournie 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] `TILE`×`TILE` 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 `TILE`×`TILE + 1` ?

6 Réduction

Il s'agit de calculer la somme des éléments d'un tableau. L'idée de base est que chaque workgroup calcule la somme de ses éléments et place son résultat dans un tableau annexe. Une fois le tableau annexe complété on itère le processus en relançant le noyau jusqu'à obtenir le résultat. Le profil du noyau est le suivant :

```
__kernel void reduction(__global float *vec, int debut, int fin)
```

2. la valeur `TILE` récupérée en ligne de commande est automatiquement transmise au noyau OpenCL sous forme d'une constante lors de la compilation.

Pour simplifier l'exercice nous utilisons en effet un seul tableau `vec` et par convention les éléments dont on doit faire la somme auront leurs indices dans `[debut,fin[`. Chaque workgroup écrira sa somme partielle à l'indice `fin + get_group_id(0)`. Au final le résultat sera placé dans `vec[fin]` par le dernier appel au noyau.
Dans un premier temps on suppose qu'on lance autant de threads que d'éléments.

1. Écrire le code du noyau `reduction`. Exemple d'algorithme : chaque thread range son élément dans d'un tableau en mémoire locale (dans sa case), puis seule la première moitié des threads continue le calcul en ajoutant à « sa case » la valeur d'un élément d'un thread devenu inactif. On itére ce processus pour obtenir le résultat.
2. Vérifier la qualité du résultat obtenu sur un un seul workgroup : au départ on peut considérer un vecteur de 8 éléments puis on augmente le nombre d'éléments.
3. Mettre en place le lancement itératif du noyau. On remarquera que le nombre d'éléments calculés par un appel au noyau est ici égal au nombre de workgroup lancés (c'est le nombre de threads divisé par la taille unitaire d'un workgroup). Il faudra faire en sorte que le nombre de threads ne dépasse pas le nombre d'éléments. Encore une fois on pourra commencer

On désire maintenant pouvoir lancé moins de threads qu'il n'y a d'éléments. Pour cela il suffit de faire en sorte que, dans un premier temps, chaque threads fasse le cumul des éléments congru à leur identité globale modulo le nombre total de threads. Il fois cette somme réalisée on peut enchaîner sur la technique de calcul précédente.
Faire quelques expériences en jouant sur le nombre de threads via l'option `-t` (qui doit apparaître après l'option `-s` pare exemple :
`./Vector -s 64m -t 1m 128`
Quelle est la meilleure combinaison pour traiter 64M éléments ?