Calcul vectoriel sur GPU : OpenCL

Il s'agit de s'initier au calcul vectoriel grâce à OpenCL qui permet de programmer les cartes graphiques de la salle 008.
Vous trouverez des ressources utiles dans le répertoire `/net/cremi/rnamyst/etudiants/opencl/`.

1 Matériel

Lancez
`LD_LIBRARY_PATH=/opt/local/cuda/lib64/ /net/cremi/sathibau/hwloc-cuda/utils/lstopo -.txt --whole-acc`
pour avoir le détail de la carte NVIDIA que vous allez utiliser. Chaque petit carré est un cœur, vous pouvez enlever l'option `--whole-accelerators` pour éviter d'avoir à les compter.

2 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

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`

En cas de doute sur le prototype ou le comportement d'une fonction, on pourra se reporter au guide de programmation OpenCL accessible au même endroit :

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

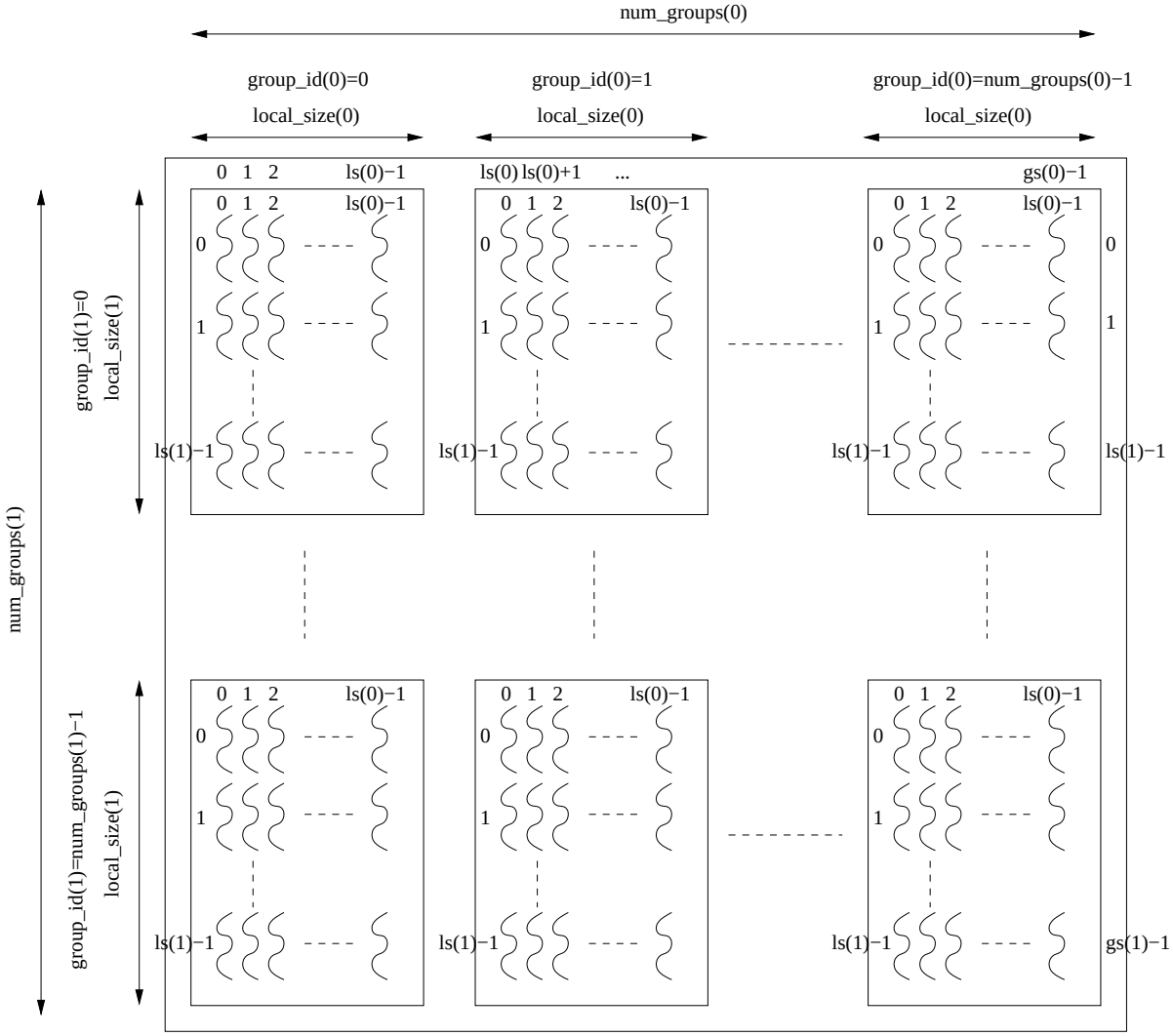
Lorsqu'on execute un « noyau » sur une carte graphique, il faut indiquer combien de threads on veut créer selon chaque dimension (les problèmes peuvent s'exprimer selon 1, 2 ou 3 dimensions), et de quelle manière on souhaite regrouper ces threads au sein de *workgroups*. Les threads d'un même workgroup peuvent partager de la mémoire locale, ce qui n'est pas possible entre threads de workgroups différents. S'il faut bien veiller à créer un grand nombre de threads pour recouvrir les temps d'accès à la mémoire, il faut aussi veiller à ne pas constituer de workgroups trop gros en nombre de threads ni en mémoire locale exigée.
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

1. En tout cas, pas de manière efficace

workgroups :

`get_global_size(d)` : taille globale de la grille selon la d^{ieme} dimension
`get_global_id(d)` : position absolue du thread courant selon la d^{ieme} dimension (entre 0 et `get_global_size(d)-1`, donc)
`get_num_groups(d)` : nombre de workgroups dans la grille selon la d^{ieme} dimension
`get_group_id(d)` : position du workgroup courant selon la d^{ieme} dimension (entre 0 et `get_num_groups(d)-1`, donc)
`get_local_size(d)` : taille du workgroup selon la d^{ieme} dimenseion. `get_local_id(d)` : position relative du thread à l'intérieur du workgroup courant selon la d^{ieme} dimension (entre 0 et `get_local_size(d)-1`, donc)

Le dessin suivant montre ceci de manière visuelle.



3 Addition de matrices

Le programme `addmat.cl` est un exemple simple de programme OpenCL effectuant une addition de matrices. Oui, le code de chaque thread est trivial : il ne s'occupe que d'une addition ! Pour comprendre comment toute l'addition est effectuée, il faut aller voir le code de `addmat.c`.
Lisez attentivement le fichier `addmat.c` et examinez comment sont détectées les cartes graphiques disponibles, comment le « noyau » destiné à s'exécuter sur la carte est compilé, comment est transféré la matrice et enfin comment l'exécution du noyau est lancée.

Remarquez, dans le programme `addmat.c`, comment le nombre de threads et les dimensions des workgroups sont fixées. Ici, on fait donc des workgroups contenant chacun `dim * 1` threads. Examinez soigneusement le calcul des indices dans `addmat.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 travaillent en fait ensemble par paquets de 32 (appelés *warp*) : ils lisent ensemble en mémoire (lecture dite *coalescée*) et calculent exactement de la même façon.
Dans le cas des GPU, on appelle souvent (à tort) *speedup* le rapport du temps nécessaire sur CPU et celui sur GPU. `add_mat` vous l'indique. Tracez une courbe du *speedup* obtenu en fonction de la valeur de `dim` (que vous pouvez passer en paramètre au programme, et utilisez une boucle `for` en shell). Utilisez `set logscale x 2` pour que ce soit plus lisible. Essayez d'utiliser des workgroups de `dim * 2` threads. Est-ce intéressant ici ? Pourquoi ?
Regardez le calcul de la « bande passante utilisée synthétique » (en Go/s), pourquoi le calcule-on ainsi ? Trouvez la bande passante interne théorique de la carte sur Internet, comparez.

4 Inversion par morceaux des éléments d'un vecteur

Le programme `vector` implémente le traitement vu en cours qui consiste à inverser l'ordre des éléments au sein de chaque tranche de 16 éléments.
Regardez le code source du noyau dans `vector.cl`. Modifiez-le pour ne pas utiliser de mémoire locale et accéder uniquement à la mémoire globale du GPU.
Faites ensuite une seconde modification afin de permuter les tranches de vecteur deux-à-deux en plus de l'inversion.

5 Transposition de matrice : à vous de jouer !

Recopiez le répertoire `AddMult` dans un nouveau répertoire `Transpose` et modifiez le programme afin de calculer la transposée d'une matrice. Par rapport au programme `AddMult`, le nouveau noyau utilisera un argument de moins, car il prendra une matrice en entrée et fournira une matrice en sortie. Il s'agit « simplement » de calculer `B[i][j] = A[j][i]`.
Donnez une première version naïve manipulant directement la mémoire globale.
Puis, en utilisant un tampon de taille `dim x dim` en mémoire locale au sein de chaque workgroup, arrangez-vous pour que les lectures ET les écritures mémoire soient correctement coalescées.

6 Propagation de la chaleur

Le répertoire `Heat` contient une version volontairement très simpliste d'une simulation de propagation de chaleur, en 1D. Regardez la version CPU `heat()` : on effectue simplement une moyenne pondérée. Pour simplifier, on ignore les problèmes de bord.
Pourquoi pour la comparaison des résultats on ne compare pas simplement avec `==` ?
Vous remarquerez que les performances ne sont pas terribles. Tracez une courbe du *speedup* obtenu en fonction du nombre de threads par block (dim).
Le problème est que chaque thread effectue plusieurs accès mémoire qui ne sont du coup pas du tout alignés. Élaborez une version utilisant un tampon partagé au sein de chaque workgroup, de façon à réaliser un miminum de lectures/écritures depuis/vers la mémoire globale de la carte.