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/opencl/`.

Pour les TP et projets nous utiliserons prioritairement les cartes graphiques de Nvidia kepler K2000 de la salle 203. Cette carte est équipée de deux multipocesseurs de 192 cœurs chacun, voici quelques une de ses caractéristiques :

Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Maximum Thread Block Size	1024

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

```
prog { options } <tile>
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 tile threads (tile × tile 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.

1. En tout cas, pas de manière efficace

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`

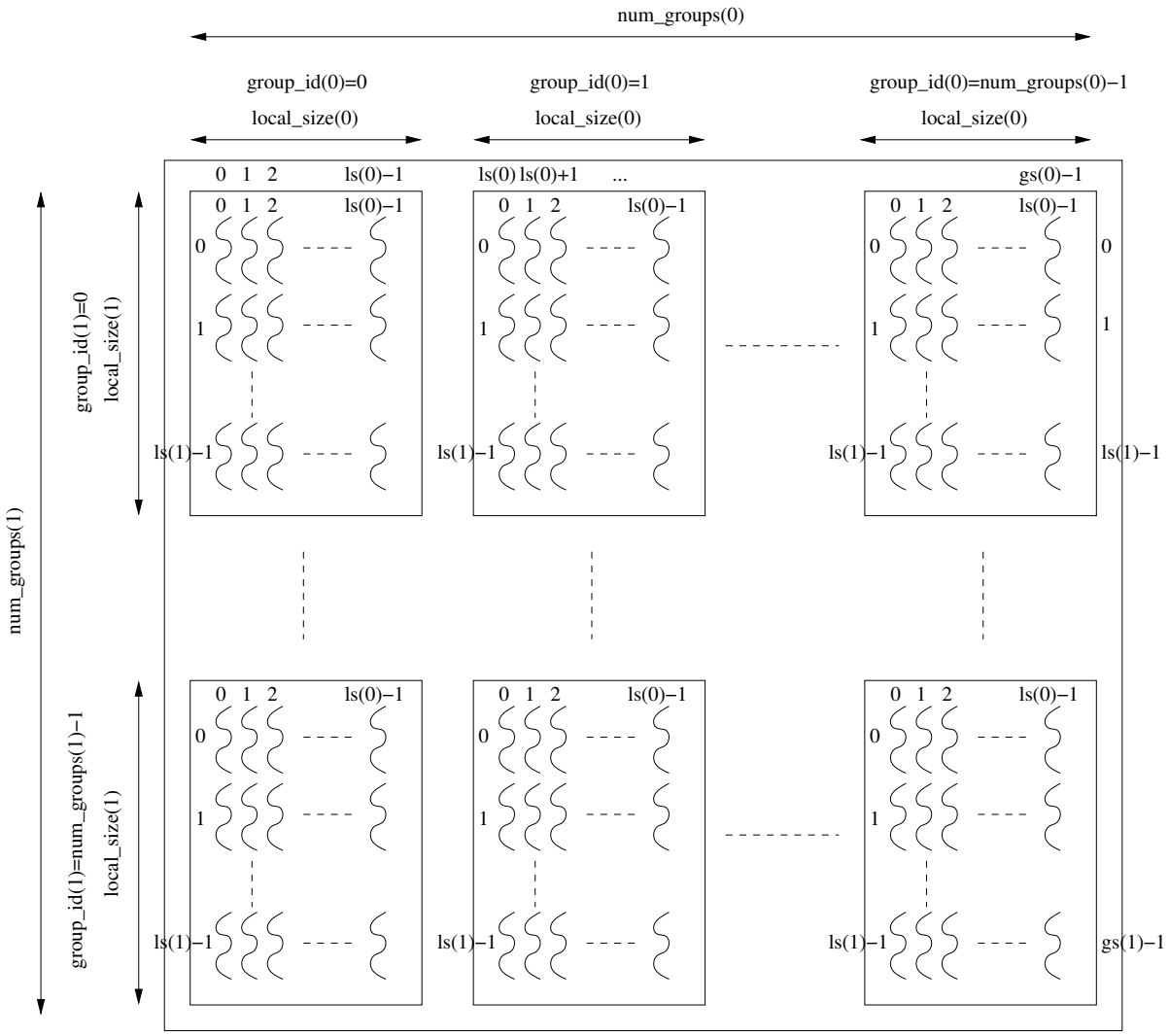
En cas de doute sur le prototype ou le comportement d'une fonction, on pourra se reporter au guide de programmation OpenCL accessible au même endroit :

`/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 *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 absolu 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

Le programme `Vector` implémente le produit d'un vecteur par un scalaire en OpenCL.
Lisez attentivement le fichier `vector.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 le vecteur et enfin comment l'exécution du noyau est lancée.
Regardez le code source du noyau dans `vector.cl`. Modifiez-le 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.

3 Addition de matrices

Le programme `AddMat` 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.cl`. 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 `TILE` × `TILE` 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 `TILE` (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. Modifiez le programme afin d'utiliser des workgroups pas forcéments « carrés », par exemple (TILE $\times$ 2, $\frac{\text{TILE}}{2}$).
Ajouter du code pour calculer la « bande passante utilisée synthétique » (en Go/s), c'est-à-dire le nombre d'octets lus ou écrits en mémoire globale par unité de temps. Comparez à la capacité théorique de la mémoire de la carte NVIDIA K2000.

4 Inversion par morceaux des éléments d'un vecteur

Le programme `SwapVector` implémente un traitement sur un vecteur qui consiste à inverser l'ordre des éléments (effet miroir).
Regardez le code source du noyau dans `vector.cl`. Augmentez la taille du *workgroup OpenCL*. Note : la valeur `TILE` récupérée en ligne de commande est automatiquement transmise au noyau OpenCL sous forme d'une constante lors de la compilation. Observez les résultats.
Modifiez le noyau pour utiliser un tableau local partagé par tous les threads d'un groupe, afin de maintenir des lectures et écritures en mémoire globale qui soient parfaitement alignées.

5 Transposition de matrice : à vous de jouer !

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 naïve manipulant directement la mémoire globale.
En utilisant un tampon de taille `TILE`$\times$`TILE` 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 en 1D

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 `==` ?
Tracez une courbe du *speedup* obtenu en fonction du nombre de threads par block (`TILE`). 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.

7 Stencil2D

Le répertoire `Stencil2D` contient une version simple d'un calcul de type « *stencil* » sur une grille à deux dimensions. Observer la version CPU (fonction *stencil*) ainsi que la version GPU (fichier *stencil.cl*) : il s'agit de calculer une moyenne pondérée de la valeur des voisins.
Une version plus optimisée consiste à utiliser un tampon local pour précharger une « tuile » de la matrice, et ensuite effectuer les calculs en minimisant les accès mémoires globaux. À vous de jouer !