Calcul vectoriel sur GPU : CUDA

Il s'agit de s'initier au calcul vectoriel grâce à CUDA qui permet de programmer les cartes graphiques de la salle 203.
Pour lancer un programme, utilisez `make run TORUN=leprogramme`

1 Découverte

CUDA est une extension du langage C permettant d'écrire des programmes s'exécutant sur une (ou plusieurs) cartes graphiques. CUDA introduit un certain nombre de qualificateurs parmi lesquels :
- `__device__` permet de déclarer une fonction exécutée sur la carte et ne pouvant être appelée que depuis la carte
- `__global__` pour les fonctions s'exécutant sur la carte mais ne pouvant être appelées que depuis "l'hôte"

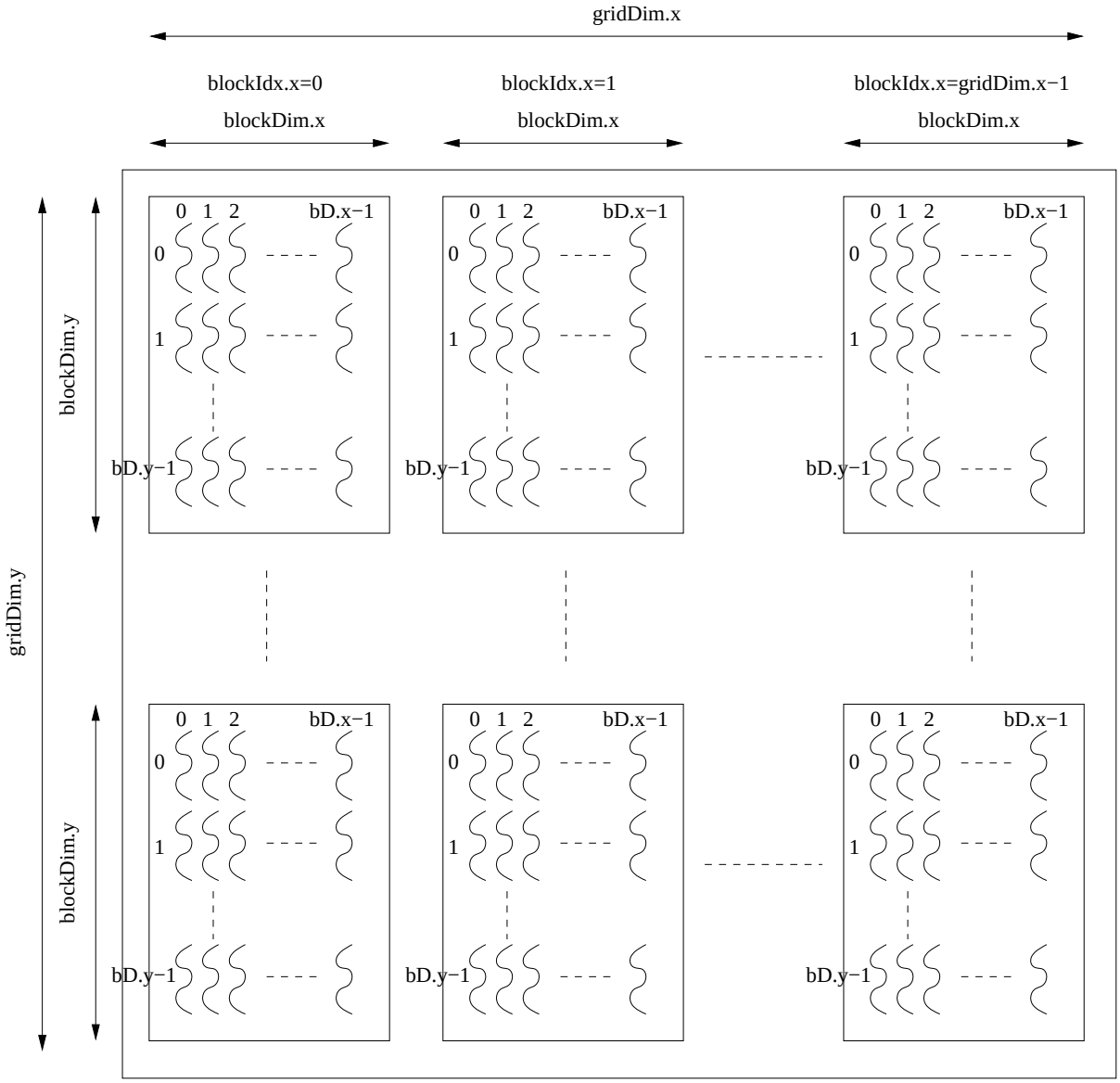
La carte graphique ne peut pas accéder à la mémoire du processeur, il faut donc transférer les données dans la mémoire de la carte avant de commencer un travail. La manipulation (allocation, libération, etc.) de la mémoire de la carte se fait par des fonctions spéciales exécutées depuis l'hôte :
- `cudaMalloc(void** ptr, size_t count);`
- `cudaFree(void* ptr);`
- `cudaMemset(void* ptr, int value, size_t count);`
- `cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind);`
  où `kind` peut être
  - `cudaMemcpyHostToDevice` (depuis la mémoire centrale vers la mémoire du GPU)
  - `cudaMemcpyDeviceToHost` (depuis le GPU vers la mémoire centrale).
  - `cudaMemcpyHostToHost` (depuis mémoire centrale vers elle-même)
  - `cudaMemcpyDeviceToDevice` (du GPU vers le GPU lui-même)

En cas de doute sur le prototype ou le comportement d'une fonction, on pourra se reporter au guide de programmation CUDA accessible à l'adresse suivante : `http://developer.nvidia.com/object/cuda.html`

Le travail à effectuer par la carte est décomposé en une matrice de `blocks` appelée grille, chaque `block` étant décomposé en une matrice de threads. Les threads d'un même bloc effectuent les même instructions et il convient de créer un grand nombre de threads pour recouvrir les temps d'accès à la mémoire.

À l'intérieur d'une fonction exécutée par le GPU, des variables sont définies afin de connaître la position d'un thread ou la dimension des blocks :
`gridDim` : dimension de la grille (nombre de blocks)
`blockDim` : dimension du block (nombre de threads)
`blockIdx` : position du block courant (à l'intérieur de la grille)
`threadIdx` : position du thread courant (à l'intérieur du block)

Le dessin de la page suivante montre ceci de manière visuelle.



Le programme `add_mat.cu` est un exemple simple de programme CUDA 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 l'appel à `add_mat_gpu` et notamment les variables `grid_dim` et `block_dim` qui permettent de choisir le nombre de blocs et le nombre de threads dans chaque bloc. Ici, on fait donc des blocks contenant chacun `dim` * 1 threads. Examinez soigneusement le calcul de `xIndex` et `yIndex` dans `add_mat_gpu`.

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 blocks de `block_dim.x = dim` sur `block_dim.y = 2` threads (il faudra bien sûr corriger à la fois le calcul de `grid_dim`, de `block_dim`, et de l'indice `yIndex`), 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.

2 Propagation de la chaleur

`chaleur.cu` 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. Essayez la version `chaleur_shared` qui utilise un tampon avec le qualificateur `__shared__`, c'est-à-dire qu'il est dans un mémoire rapide du GPU, partagée entre les threads d'un même bloc. Voyez combien le résultat est bien meilleur. Lisez le code est comprenez pourquoi cela effectue bien le bon calcul. Avez-vous une idée de pourquoi le *speedup* est bien plus grand que pour l'addition ? (faites varier la taille des blocs).

3 À vous de jouer !

`transposition.cu` contient une version triviale de la transposition. Vous remarquerez que les performances sont juste « bonnes » (faites varier la taille des blocs et tracez une courbe). C'est parce que certes les lectures mémoire sont bien coalescées, mais les écritures mémoire ne le sont pas. Et inversement si l'on inverse les indices !
Une solution est donc de passer par un petit tableau partagé de dimension `DIM x DIM`, dans lequel les threads stockent ce qui est lu, puis font la transposition vers un autre petit tableau partagé (pas de problème de lecture/écriture coalescée ici puisque c'est de la mémoire locale), et enfin recopient le résultat dans la matrice (attention, au bon endroit !). Essayez, testez, mesurez (faites varier la taille des blocs). Vous pouvez même effectuer le calcul en seulement deux opérations, via un seul tableau partagé.