TD GPGPU n°2

GPGPU Courses
MII 2 option Sciences de l'Image et IMAC 3 option Prog 3D

Mémoire constante et textures

Du bon usage des espaces mémoires dans les architectures GPU

Dans ce TP, vous allez utiliser deux nouvelles zones mémoires de vos GPUs : la mémoire constante et la mémoire de textures. Une nouvelle fois, ne codez que dans les fichiers `student`.

Zone de mémoire constante

Comme son nom l'indique, cette zone mémoire est utilisée pour stocker des données qui ne seront pas modifiées durant l'exécution du noyau. Autrement dit, les données chargées en mémoire constante ne seront accessibles qu'en lecture. CUDA vient avec 64Ko de mémoire constante optimisée pour un accès en lecture. Ainsi, l'utilisation de cette zone mémoire peut augmenter les performances d'un programme en réduisant la bande passante nécessaire à son exécution.

La mémoire constante est déclarée en utilisant le mot-clé `_constant_`. Notez que cette zone mémoire est allouée statiquement. Il n'est donc pas nécessaire de se préoccuper de son allocation (`cudaMalloc`) ou de sa libération (`cudaFree`). Le principal changement se trouve dans l'initialisation des données. Au lieu d'utiliser `cudaMemcpy` comme pour la mémoire globale, il faut utiliser `cudaMemcpyToSymbol`.

Mémoire de textures

La mémoire de textures est particulièrement adaptée pour les traitements ayant une forte localité spatiale. Contrairement à la mémoire constante, cette zone mémoire est « cachée » et peut donc considérablement améliorer les performances de vos algorithmes. Toutefois, en cas de défaut de cache, les performances peuvent être inversement impactées. Une texture est déclarée ainsi : `texture<type,dim,mode>`. Les trois attributs correspondent à :

- Un type : `float`, `int2`, `float4`, etc.
- La dimension de la texture : 1, 2 ou 3
- Le mode de lecture : `cudaReadModeNormalizedFloat` ou `cudaReadModeElementType`.

Avant d'être utilisée, une texture doit être déclarée et liée à une zone allouée de la mémoire globale (*binding*). Une fois utilisée, la texture doit bien sûr être déliée. Notez que depuis les cartes graphiques d'architecture Kepler de Nvidia, CUDA 5.0 introduit les *texture objects* ou *bindless textures*.

Convolution d'images

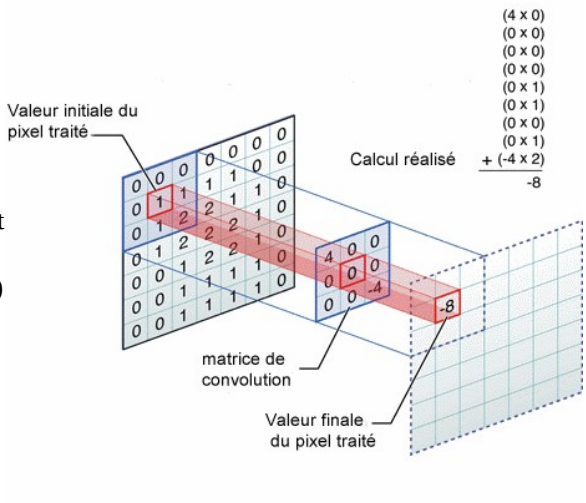
En traitement d'image, la convolution consiste à appliquer un filtre sur une image afin de faire apparaître des effets ou de récupérer des informations : flouter une image, détecter les contours, etc.

À chaque filtre est associé une matrice qui sera appliquée à chaque pixel de l'image tel que :

$$f_{res} = (f \cdot h)[x, y] = \sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} h[i, j] \cdot f[x - i, y - j]$$

où :

- f[x, y] correspond à la valeur du pixel initial (entre 0 et 255) aux coordonnées x et y ;
- f_{res} [x, y] est la valeur du pixel résultat (entre 0 et 255) aux coordonnées x et y ;
- h : matrice de convolution de taille $[x_1 , x_2]\times[y_1 , y_2]$.



Les TD ont été écrit très majoritairement par Maxime Maria (maintenant Maître de Conférences au XLIM à Limoges)

Dans ce TP, vous devez paralléliser ce traitement sur GPU. Quatre matrices de convolution sont codées et utilisables via l'argument de programme `-c <C>`, où `<C>` correspond au numéro de type de convolution :

Image source | 0 : Bump (3x3) | 1 : Sharpen (5x5) | 2 : Edge detection (7x7) | 3 : Motion blur (15x15)

N.B. : Pour chaque exercice, analysez les performances données par l'application pour plusieurs tailles de matrices de convolution et avec différentes images.

N.B. 2 : Dans ce TP, vous allez implémenter plusieurs versions du même traitement, gardez le code de chaque version, l'objectif est de les comparer entre elles (et que je puisse voir toutes les versions) ! Évitez de laisser des versions en commentaires. **En bref, écrivez une fonction par version.**

Exercice 1 : Première version « naïve »

Implémentez une première version CUDA permettant d'appliquer la matrice de convolution à chaque pixel de l'image. Vous pouvez vous inspirer de la version CPU (`convCPU`) proposée dans le fichier `main.cu`.

Exercice 2 : Avec mémoire constante

Passez la matrice de convolution en mémoire constante (rappel : conservez la version précédente !).

Exercice 3 : Avec la mémoire de texture 1D

Passez l'image source en mémoire de texture 1D.

Exercice 4 : C'est toujours mieux en 2D

Passez l'image source en mémoire de texture 2D. Indice : cherchez du côté de `cudaMallocPitch` et `cudaMemcpy2D`.