

Partie 2 : calcul sur GPU

Exercice 1 Première prise en main des GPU. Tous les fichiers mentionnés sont disponibles sur moodle.

1. Récupérer le fichier `example1.cu`, compiler le avec `nvcc` et exécuter le programme généré.
2. Récupérer le fichier `deviceQuery.cu` et compiler le avec `nvcc`. Attention, il vous faudra ajouter l'option de compilation `-I/usr/local/cuda-11.5/samples/common/inc`. Exécuter le programme généré et regarder les caractéristiques CUDA de votre carte graphique (taille des grilles, taille des blocs, nombre maximal de thread par bloc, ...).
3. Récupérer le fichier `example2.cu` et compiler le avec `nvcc`. Lors de son exécution ce programme doit afficher les entiers de 0 à 9. Lancer l'utilitaire `cuda-memcheck` pour vérifier votre programme (`cuda-memcheck ./monprog`). Recompiler votre fichier en ajoutant les options de compilation `-Xcompiler -rdynamic -lineinfo`. Relancer `cuda-memcheck` et corriger le kernel GPU.

Exercice 2 On souhaite implanter un kernel CUDA pour faire le calcul sur GPU de la somme de deux vecteurs de taille n contenant des `float`. Notre kernel exécutera au moins n threads qui s'occuperont de calculer une entrée du résultat.

Nous allons considérer que notre kernel sera exécuté sur une grille et des blocs ayant une seule dimension car il n'y a pas lieu à utiliser de dimension supérieure. La règle avec les architecture CUDA est que le nombre de threads dans un bloc doit être un multiple de 32 (la taille d'un *warp*). Nous allons dans un premier temps nous intéresser à déterminer quelles sont les paramètres possibles d'exécution de notre kernel CUDA.

1. Donner l'ensemble des configurations possibles pour les tailles de grille et de blocs pour des vecteurs de taille 1024, 1023 et 1025.
2. Quelle est la taille de grille lorsqu'on souhaite des blocs de taille 128 avec des vecteurs de taille n ?
3. Caractériser les valeurs de n pour lesquelles notre kernel laissera des threads inactifs ?

Nous allons maintenant programmer et exécuter ce kernel.

5. Compléter le kernel ci-dessous pour qu'il fasse l'addition sur GPU de vecteur de `float` ($C = A + B$). Pour rappel : chaque entrée du résultat doit être calculée par un thread différent (le nombre de thread lancé sera $\geq n$) ; on peut récupérer l'indice d'un thread dans un bloc au travers de la variable globale `threadIdx`, l'indice du bloc au travers de la variable globale `blockIdx` et la taille du bloc au travers de la variable globale `blockDim`. **Attention** : chacune de ces variables contient des attributs `x`, `y` et `z` pour récupérer la valeur de la dimension concernée ; il faudra également tenir compte que certains threads ne serviront à rien.

```

1  __global__ void vectorAddition(float* C, const float* A, const float* B, size_t n){
2      ...
3  }
```

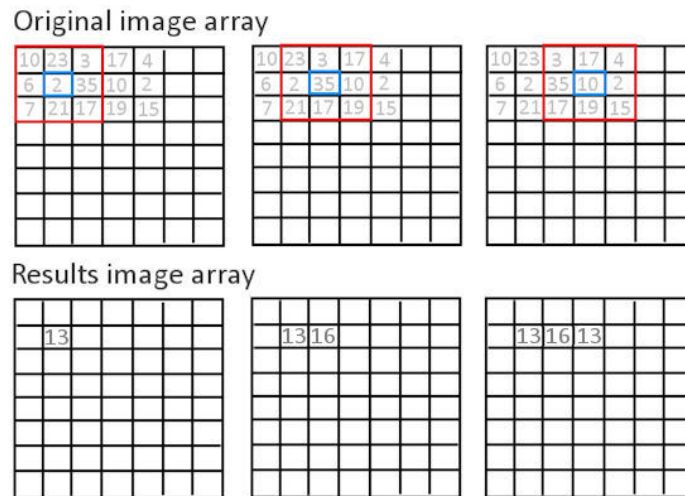
6. Écrire une fonction classique en C++ qui fait le calcul de la somme de deux vecteur de float (sans CUDA). Écrire un programme qui génère deux tableaux `hA` et `hB` contenant n flottants initialisés avec des valeurs entières aléatoires entre 0 et 65535 et qui calcule leur somme dans un vecteur `hC`. La valeur de n doit être passé comme paramètre du programme.
7. Dans ce même programme, utiliser les fonction `cudaMalloc`, `cudaMemcpy`, `cudaFree` pour préparer l'exécution de votre kernel sur les données stockées dans `hA` et `hB`.
8. Utiliser votre kernel pour faire le calcul et comparer le résultat avec le vecteur `hC` obtenu par l'appel de votre fonction CPU (Attention, il y a de nombreuses copies de données à effectuer). Vous essayerez l'ensemble des configurations possibles de lancement de votre kernel en supposant que la taille des blocs et un multiple de 32.
9. En vous inspirant de l'addition de vecteur, proposer une nouveau kernel qui permet de faire l'addition de deux matrices stockées au format *RowMajor*, cf signature ci-dessous. Vous ferez comme précédemment en définissant une fonction CPU qui fait le même calcul et vous vérifierez que les résultats sont identiques. Pour ce kernel vous devrez utiliser une grille et des blocs en deux dimensions. Vous testerez votre kernel sur quelques configurations pertinentes (petite grille + grand bloc, moyenne grille+moyen bloc, grande grille + petit blocs).

- Comparez les performances de vos kernels GPU et des fonctions CPU équivalente. Pour cela, vous utiliserez votre classe `EvalPerf`. Attention, le lancement d'un kernel étant asynchrone, il faudra que l'appel à votre kernel soit suivi de l'instruction `cudaDeviceSynchronize()`; pour que le temps d'exécution soit correct.

Exercice 3 Nous souhaitons écrire un traitement d'image simple sur GPU. Pour cela, nous allons nous intéresser à un filtre moyennneur qui permet de flouter une image. Le principe du filtre moyennneur consiste à remplacer un pixel donné par la moyenne des valeurs des pixels qui l'entoure. La notion d'entourage est relié à la taille du filtre. Pour faire simple, nous considérerons un filtre de taille $2B + 1$ de telle sorte que la valeur d'un pixel $P_{x,y}$ à la position (x, y) sera remplacé par

$$\frac{1}{(2B + 1)^2} \sum_{i=x-B}^{x+B} \sum_{j=y-B}^{y+B} P_{i,j}$$

L'image ci-dessous illustre le concept de ce filtre pour $B = 1$.



Bien entendu, pour les pixels en bordure la taille du filtre s'adaptera pour ne contenir que les pixels existants. Autrement dit, on ne considère que les voisins existants du pixel $P_{x,y}$ qui sont à une distance maximale de B en abscisse et en ordonnée.

Pour vous simplifier la gestion de la lecture/écriture de fichier d'image, vous trouverez sur l'ENT un fichier `image-pnm.h` qui vous fournit une classe `Image`. Cette dernière permet de manipuler des images au format PNM ASCII : PBM (encodage monochrome avec pixel noir ou blanc), PGM (encodage monochrome avec 256 niveau de gris) et PPM (encodage couleur RGB avec 256 niveau de couleur par canal). Vous trouverez également un fichier d'exemple `copy-image.cpp` qui crée une copie d'une image à l'un des formats supporté.

- Écrire une fonction `blur_pbm` qui permet d'appliquer un filtre moyennneur sur une image. Pour vous simplifier la tâche votre fonction sera de type `void` et prendra en paramètre une image source et une image cible. Seule l'image cible sera modifiée. Vous utiliserez un paramètre global pour définir la valeur de B dans la taille du filtre moyennneur (par exemple $B=4$).
- En vous inspirant de votre code précédent, proposer un kernel GPU qui fait la même tâche. ATTENTION, ce kernel ne devra pas manipuler la classe image, par conséquent votre kernel devra avoir la signature suivante :

```
--global__ blur_pbm_kernel(uint8_t *ImgOut, const uint8_t* imgIn, int width, int height);
```
- En vous inspirant du fichier d'exemple `copy-image.cpp`, écrire un fichier CUDA (`blur-image.cu`) qui permet de lire un fichier image au format PBM et qui génère un nouveau fichier au format PBM dans lequel le filtre moyennneur a été appliqué. Vous trouverez des images sur l'ENT pour tester votre programme.
- Évaluer les performances de vos codes et essayer de trouver de manière empirique quelle est la meilleure configuration de lancement de votre kernel GPU.
- À votre convenance, proposer un autre kernel de traitement d'image et tester le sur les images proposées. Si vous n'avez pas d'idée, vous pouvez soit faire un kernel de passage d'une image

couleur en une image à niveau de gris, soit faire une rotation de 180 degrés, soit un flip horizontal de l'image.

Exercice 4 Nous allons reprendre vos codes de produit de matrices et voir si vous êtes capable d'être plus rapide avec des calculs sur GPU.

Le principe d'un produit de matrice « naïf » sur GPU consiste à dire que chaque thread s'occupe de calculer une unique entrée de la matrice résultat. Si l'on multiplie des matrices carrées de dimension n , cela implique donc de lancer au moins n^2 threads. Pour coller au plus près avec les matrices, nous considérerons que notre kernel utiliser des grilles et des blocs deux dimensions.

1. Écrire un kernel GPU qui calcule un produit de matrice carrée contenant des float. Bien entendu, ce kernel doit prendre en compte que certains thread ne feront aucun calcul.
2. Modifier votre programme pour qu'il puisse exécuter en plus de vos fonctions CPU, ce nouveau kernel GPU. Vous comparerez les performances de vos codes sur des matrices aléatoires de taille 2^k pour k allant de 8 à 12. Vous essaieriez de trouver de manière empirique quel est le meilleur choix pour le nombre de threads par bloc.