Travaux Pratique 6

Ce sujet est en lien avec le quatrième chapitre du cours, et concerne la programmation CUDA. Les mêmes commentaires que ceux des derniers TP s'appliquent ici aussi.

Dans cette séance, l'objectif est de pratiquer la programmation CUDA, autour des patrons en temps constants (sur machine PRAM). Le premier exercice est un MAP unaire proche du cours (filtre Sépia), et ne devrait pas poser de difficulté. Le second exercice est un MAP binaire, avec un foncteur amusant transformant une image, qui sera ensuite utilisé dans les deux exercices suivants qui consistent à déplacer des morceaux d'images (permutations). Le dernier exercice, légèrement plus difficile, consistera à utiliser les contraintes matérielles d'un GPU au mieux, c'est-à-dire ici le principe d'accès coalescents à la mémoire globale.

ATTENTION : pour rappel, tous les travaux non rendus sont transformés en note 0. Si un tiers ou plus des TP ne sont pas rendus (à partir de 4 TP), la note attribuée à l'épreuve sera ABI sans contrôle de rattrapage, conduisant de fait à l'échec au module et à l'année.

> Votre travail est à rendre (il le sera à chaque fois) sous la forme du code (et uniquement la partie « student ») accompagnée d'un rapport au format PDF, le tout dans une archive compressée au format ZIP. Ne respecter pas ces contraintes, et votre note en sera diminuée de quelques points.
>
> **Vous ne devez modifier que ce qui se trouve dans le répertoire « `./student/` » !**
>
> Notez que chaque exercice vient avec un squelette, qui s'occupe de la ligne de commande, du lancement de votre code (défini dans une classe particulière), et d'une vérification sommaire du résultat (lorsque c'est possible). Utilisez l'option « `-h` » ou « `--help` » pour connaître le fonctionnement de la ligne de commande …

Exercice 1

L'objectif est d'écrire une fonction réalisant un MAP unaire (fichier `UnaryMap.h`). L'application est un filtrage SEPIA d'une image. La fonction unaire est donnée (celle réalisant la transformation d'un pixel RGB vers du SEPIA).

Attention : la taille des tableaux n'est pas toujours un multiple du nombre de threads par bloc. Conséquence : il faut vérifier la pertinence de l'écriture dans le noyau, via un garde-fou … Votre version sera implémentée dans une fonction réutilisable, utilisant donc un foncteur (donné pour l'exercice).

Exercice 2

Ici aussi il faut écrire un MAP, mais binaire ce coup-ci (fichier `BinaryMap.h`). Notez que son utilisation dans l'exercice utilise un `DeviceBuffer` à gauche et un foncteur à droite… La fonction à appliquer retourne l'élément à gauche si celui à droite vaut False, sinon une couleur donnée.

Le tout produit un effet *vignette (cf. page 3)*. Cet effet consiste à ajouter un bord à chacun des 3×3 blocs de l'image.

NB : **pensez parallèle** ! Un algorithme séquentiel est inadaptable ...

Exercice 3

Implémentez le patron GATHER.

Exercice 4

Implémentez le patron SCATTER.

Exercice 5

Dans ce dernier exercice, il vous est demandé de tenir compte du mode d'accès à la mémoire globale depuis chaque **warp** (coalescence). L'idée est de filtrer une image (cf. page 5). Le filtre est une fonction fournie sous la forme d'un tableau de coefficients à appliquer pour calculer la valeur de chaque pixel de l'image produite. Il s'applique sur les pixels du voisinage du pixel de même position dans l'image source. Le filtre s'applique sur une grille carrée des voisins du pixel à filtrer, avec une taille impaire (*e.g.* $3 \times 3,\ 5 \times 5,\ 7 \times 7,\ \ldots$). Par exemple, si le filtre est de taille 3×3, alors le calcul du pixel de position (x, y) sera par composante R, G et B :

$$D(x,y) = \sum_{i=0}^{2} \sum_{j=0}^{2} F(i,j) \times I(x+i-1, y+j-1),$$

avec D l'image destination (celle à écrire), I l'image source et F la fonction à appliquer. Notez que les « -1 » dans les indices de l'images sources viennent de la taille du filtre (dans cet exemple). En général il faut utiliser $size/2$. Si $size$ est 3, vous obtenez 1 ! Lorsque $size$ vaut 5, le résultat est 2 ; pour $size = 7$ vous aurez 3, *etc*.

Notez que les pixels « manquants » (proche du bord) sont obtenus en repliant l'image sur elle-même (symétrie axiale). Par exemple le pixel `(-1,2)` deviendra par symétrie le pixel `(0,2)`. Le pixel `(-1,-2)` sera le pixel `(0,1)`. Le pixel `(width+3,height)` sera le pixel `(width-4,height-1)` puisque les derniers indices par ligne et colonne sont respectivement `width-1` et `height-1`.

Expérimentez différents schémas d'accès/répartition des calculs pour définir la taille de la grille correctement.

Résultats sur l'image « Raffael_012.ppm ».

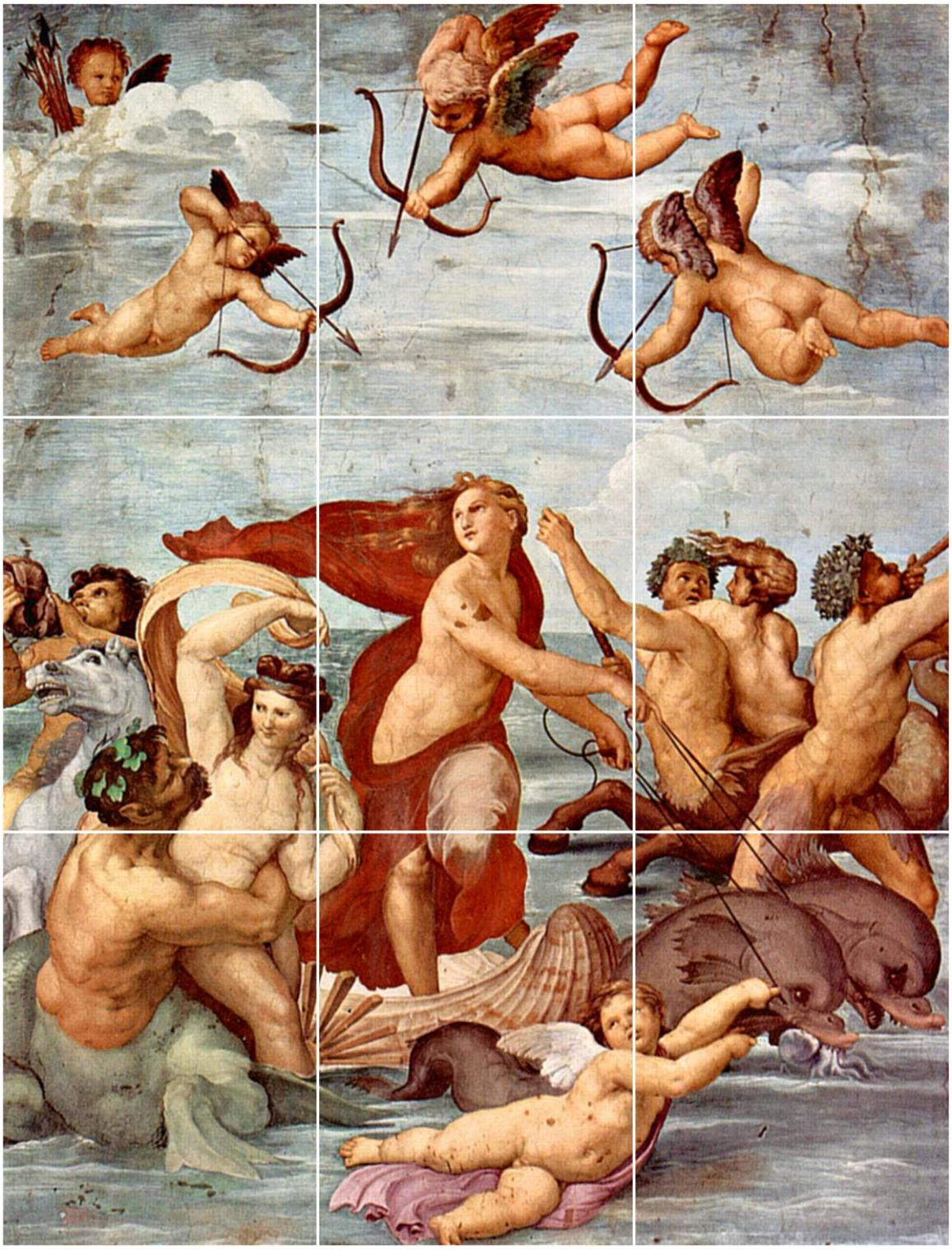

Figure 1: effet vignette (exercice 2)

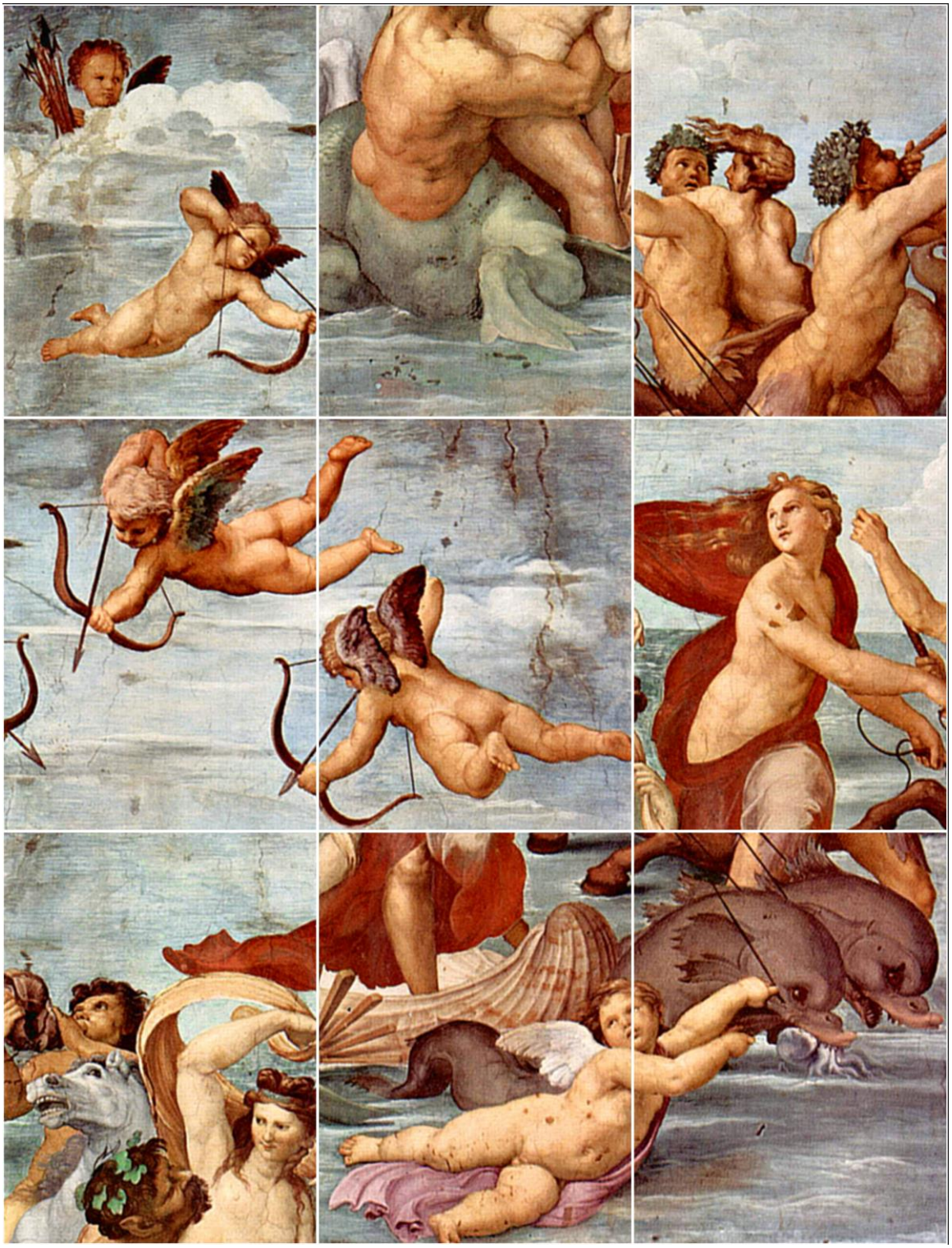

Figure 2 : résultat exercices 3 et 4 ; vignette puis permutation (attention, la permutation est aléatoire, donc chaque exécution du programme donnera un résultat différent)

Figure 3: exercice 5 ; filtre de (demi) taille 8