Travaux Pratique 7

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) et du patron REDUCE. Dans tous les exercices, l'idée reste la même : transformer une image en lui donnant un effet par bloc de taille 32 par 32. La couleur des pixels de chaque bloc sera unique par bloc dans l'image résultat. Elle résulte d'un calcul assez simple : la moyenne des pixels du même bloc de l'image source. Vous trouverez la version CPU dans la classe `src/Exercise1/ImageBlockEffect.h`.

Nous allons jouer avec l'implantation de la réduction.

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

Ce premier exercice se veut très simple : il s'agit de calculer la couleur de chaque bloc de l'image résultat. Pour cela, il faut reprendre la version de la réduction vue en cours, et accessible dans `utils/OPP/OPP_cuda_reduce.cuh`. Cela passe par un noyau qui doit :

- Charger le bloc en mémoire partagée.
- Appliquer **la réduction par bloc** en reprenant la fonction du cours.
- En déduire la couleur du bloc et l'écrire dans l'image résultat (en remplissant le bloc, donc).

Ici, chaque bloc de l'image est traité par un bloc de threads de taille 32 par 32 (soit 32 warps). Cette première version est une application directe du cours (vous pouvez quand même remplacer le foncteur par l'addition classique).

Attention à **effectuer votre réduction sur des réels** et non pas des octets !

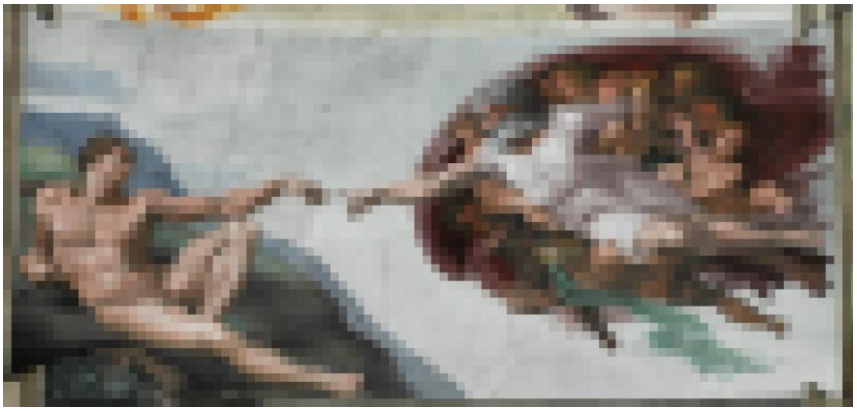


Figure 1 : effet bloc sur l'image God2-Sistine_Chapel.ppm

Exercice 2

Vous avez probablement noté que la version GPU n'est pas beaucoup plus rapide que la version CPU. Comme dit René, ce n'est pas taupe. Plusieurs raisons expliquent cela, nous allons essayer de les traiter l'une après l'autre. Dans cet exercice, l'idée est de réduire la **pression sur les registres**.

Vous vous souvenez qu'un SMP possède un nombre fixe de registres (par exemple 64k ou 128k). Ces registres sont utilisés pour stocker les variables *auto* (i.e. déclarées dans vos fonctions), ainsi que les mémoires *partagée* et *constante* (les paramètres des fonctions sont en mémoire globale, en revanche). En utilisant la mémoire partagée pour la réduction, notre première version a donc requis un grand nombre de registres par bloc. Calculons cela : il vous a fallu 1024 registres pour une étape de réduction (voire 3 fois plus si jamais vous avez chargé toutes les couleurs en même temps !). Vous avez lancé 32 fois 32 threads par bloc, donc 1024. En supposant que chaque thread nécessite 20 registres (ce qui est très très peu, en fait), alors un bloc requière 1024+20*1024 registres, soit 21k ; un SMP avec 64k registres peut alors faire tourner 3 blocs en parallèle, et pas un de plus. En supposant 32 registres maximum par thread, alors vous aurez un seul bloc par SMP ! Pas terrible pour l'efficacité !

Ainsi, comme pour la version vue en cours, nous devons réduire le nombre de threads par bloc pour optimiser le parallélisme. **Utilisez au maximum 256 dans cet exercice**. Pour cela il faudra modifier le fonctionnement de la réduction : dans la partie chargement il faut **qu'un thread charge plusieurs valeurs, tout en *respectant la propriété d'associativité*** ; autant ajouter ces valeurs dans un registre privé, puis charger cette somme dans la mémoire partagée.

Attention : un bloc de 256 threads traite toujours 32x32 pixels !

Indiquez dans le rapport ce qui fonctionne le mieux sur votre GPU (indiquez lequel) en termes de taille d'un bloc (faites des essais avec l'option `-w`, mais attention : le schéma de calcul du *reduce* ne fonctionne qu'avec un nombre de warps qui est une puissance de 2 !).

Figure 2 : image originale (wraps avocat/crevette)

Figure 3 : application de l'effet bloc (sur wraps avocat/crevette)

Exercice 3

Dans l'exercice 2, le respect de **la propriété d'associativité** a rendu le chargement de la mémoire cache un peu délicat (chargement par bloc *pour un même thread*).

Une version plus simple consiste à charger les données par bloc *pour tous les threads* : un thread charge la valeur à l'indice correspondant à sa position, puis la valeur de même indice plus le nombre total de threads, puis la valeur à son indice plus le nombre total de threads fois 2, etc. Ce schéma plus efficace (en termes de coalescence) nécessite la **propriété de commutativité**, qui n'est pas requise dans une réduction classique (pensez au produit de matrices …). Ici, l'opération associative est une simple addition sur des réels, donc elle est commutative.

Modifiez dans cet exercice le chargement des données en mémoire cache.

Exercice 4

Il est possible de tirer parti de la propriété de commutativité pour accélérer encore plus la réduction. Modifiez le schéma de calcul de la fonction `reduceJumpingStep` en ce sens, de façon à avoir moins de warps au travail de sorte à **réduire les latences dues aux synchronisations**.

- Sur une feuille de papier, dessinez le schéma de calcul actuel (avec 4 warps de 4 threads chacun pour simplifier, c'est suffisant pour réfléchir…).
- Proposez un second schéma minimisant le nombre de warps au travail, grâce à la commutation de l'addition. L'idée est d'avoir la réduction des 64 dernières valeurs dans un unique warp …
- Codez ce nouveau schéma.

NB : les dessins mentionnés ici se trouvent page 8 en version vue en cours, puis page 14 pour cet exercice dans : https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Exercice 5

La version précédente est efficace car moins de warps travaillent. Vous avez observé sur votre feuille de papier que les dernières itérations n'utilisent qu'un seul et unique warp : pour ajouter 64 valeurs, puis 32, puis 16, puis 8, puis 4, puis enfin les 2 dernières. Un seul warp pour traiter ces 6 étapes ! Nagerions-nous en plein Tolkien ? Et pourtant, comme un grand béotien vous continuez à synchroniser les threads, ce qui est inutile au sein du même warp (SIMD oblige !).

Supprimez, pour ces dernières réductions (et uniquement celles-ci !) la synchronisation. **Attention** : la mémoire partagée devra avoir l'attribut `volatile` … sinon le compilateur va chercher à optimiser avec des variables privées.

Refaites l'étude sur le nombre de threads optimal.

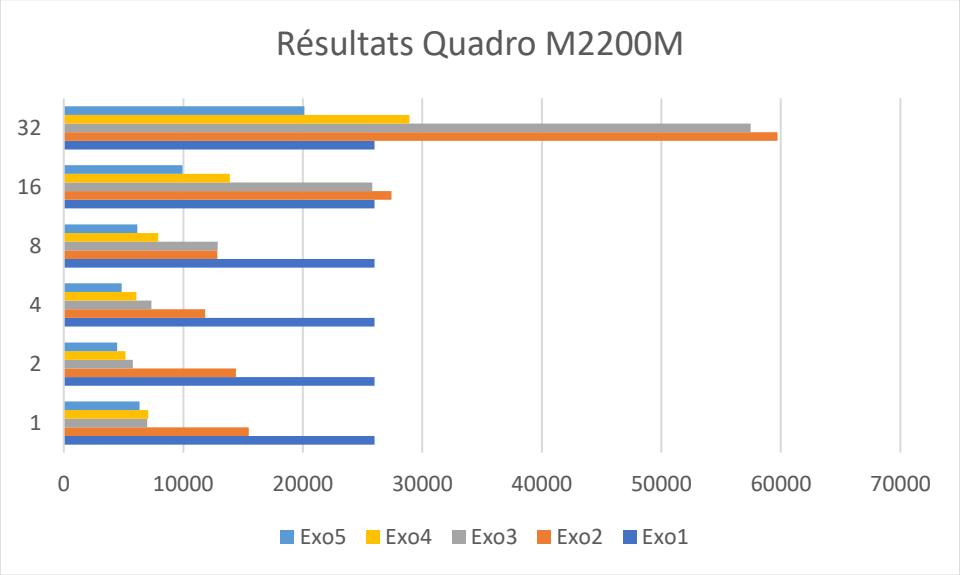
Et voilà, vous avez de quoi écrire une nouvelle version de la réduction, pour une fonction associative **et commutative** …

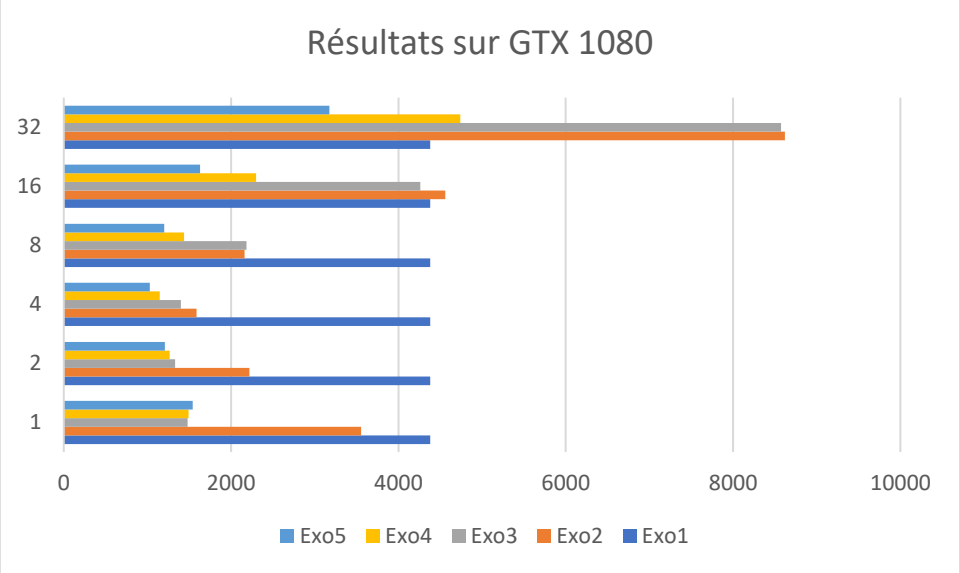
Figure 4 : image SunFlowers.ppm

Figure 5 : filtre sur SunFlowers.ppm

Résultats

Quelques résultats sur mes machines pour l'image « `SunFlowers.ppm` ». En abscisse le temps moyen en micro secondes. En ordonnée, le nombre de warps.





Vous remarquerez que le nombre de warps idéal dépend du GPU utilisé : pour l'exercice 5 l'idéal est 2 warps sur la Quadro (4457 us), et 4 sur la GTX (1027 us) …

Notez pour les plus curieux qu'il est encore possible d'accélérer ce traitement (moins de 800 us sur GTX 1080) en réduisant encore la pression sur les registres via des *templates*.