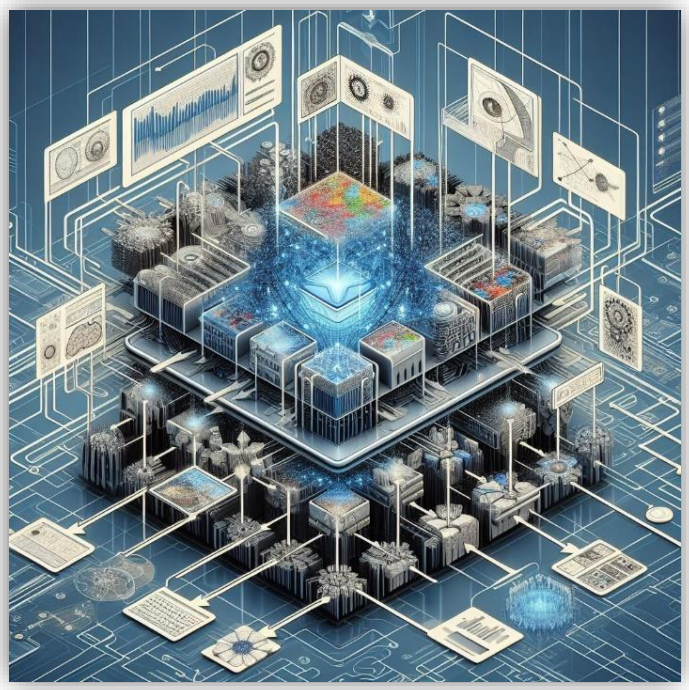

Sommaire

Spécifications de ma machine	
Processor	AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz
Installed RAM	16.0 GB (13.9 GB usable)
System type	64-bit operating system, x64-based processor
Windows	Windows 11 Home
GPU Device	NVIDIA GeForce GTX 1650

GPU-Total amount of global memory:	4096 MBytes (4 294 639 616 bytes)
GPU-Max dimension size of a grid size (x,y,z):	(2 147 483 647, 65 535, 65 535)
GPU-Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
GPU-Total number of registers available per block:	65 536
GPU-Warp size:	32

Exercice 1

Voici les résultats obtenus de l'implémentation avec des images de différentes tailles et en mesurant le temps d'exécution de la fonction rgb2hsv :

Taille des Images	Moyenne du temps d'exécution de la fonction RGB2HSV sur GPU (µs) RGB -> HSV / HSV -> RGB	Moyenne du temps d'exécution de la fonction RGB2HSV sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	131 / 56	2 421
Paris.ppm (562 000 pixels)	144 / 97	4 360
Nuit.ppm (1 764 000 pixels)	303 / 286	16 610
MonSalon.ppm (10 036 224 pixels)	1 518 / 1 379	86 522

En ce qui concerne le code, La fonction *RGB2HSV_kernel* prend en entrée une image en couleur RGB et convertit chaque pixel en espace de couleur HSV en utilisant la fonction RGB2HSV. Les résultats sont ensuite répartis dans trois tableaux distincts pour les composantes Hue, Saturation et Value. Cette répartition en trois tableaux distincts vise à optimiser le débit mémoire(la coalescence). La fonction *HSV2RGB_kernel* effectue l'opération inverse. Elle prend en entrée trois tableaux représentant les composantes Hue, Saturation et Value d'une image en espace de couleur HSV, et convertit chaque pixel en espace de couleur RGB en utilisant la fonction HSV2RGB.

En ce qui concerne les résultats, on observe que le temps d'exécution de la conversion d'images entre les espaces de couleurs RGB et HSV est nettement plus rapide sur le GPU que sur le CPU. Par exemple, pour l'image *CaravaggioUrsula.ppm*, le GPU a pris en moyenne 131 µs pour la conversion RGB vers HSV et 56 µs pour la conversion HSV vers RGB. En comparaison, le CPU a pris en moyenne 2 421 µs pour la même conversion, soit environ 18 fois plus longtemps. Pour l'image *Paris.ppm*, le GPU a pris en moyenne 144 µs pour la conversion RGB vers HSV et 97 µs pour la conversion HSV vers RGB. Le CPU a pris en moyenne 4 360 µs, soit environ 30 fois plus longtemps. On observe donc que l'écart de performance entre le GPU et le CPU augmente avec la taille de l'image. Pour l'image *Nuit.ppm*, le GPU a pris en moyenne 303 µs pour la conversion RGB vers HSV et 286 µs pour la conversion HSV vers RGB. Le CPU a pris en moyenne 16 610 µs, soit environ 55 fois plus longtemps. Pour l'image *MonSalon.ppm*, le GPU a pris en moyenne 1 518 µs pour la conversion RGB vers HSV et 1 379 µs pour la conversion HSV vers RGB. Le CPU a pris en moyenne 86 522 µs, soit environ 57 fois plus longtemps.

Ces résultats s'expliquent par le fait que le GPU est capable d'exécuter de nombreux threads en parallèle, ce qui permet de traiter chaque pixel de l'image simultanément. En revanche, le CPU traite les pixels de manière séquentielle, ce qui est beaucoup plus lent pour les grandes images.

Exercice 2

Voici les résultats obtenus de l'implémentation avec des images de différentes tailles et en mesurant le temps d'exécution du calcul de l'histogramme :

Taille des Images	Moyenne du temps d'exécution du calcul de l'histogramme sur GPU (µs)	Moyenne du temps d'exécution du calcul de l'histogramme sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	125	1 228
Paris.ppm (562 000 pixels)	135	2 438
Nuit.ppm (1 764 000 pixels)	158	7 315
MonSalon.ppm (10 036 224 pixels)	448	43 768

En ce qui concerne le code, la fonction run_Histogram prend en entrée un tableau de valeurs en float, qui sont les valeurs des pixels de l'image, et un tableau pour stocker l'histogramme résultant. Les valeurs sont comprises entre 0 et 255. Un foncteur est défini pour convertir les valeurs en float en valeurs entières non signées. Ce foncteur est ensuite utilisé pour calculer l'histogramme des valeurs avec la fonction computeHistogram.

En ce qui concerne les résultats, On observe que le temps d'exécution du calcul de l'histogramme est nettement plus rapide sur le GPU que sur le CPU. Par exemple, Pour l'image *CaravaggioUrsula.ppm*, le GPU a pris en moyenne 125 µs pour le calcul de l'histogramme. En comparaison, le CPU a pris en moyenne 1 228 µs, soit environ 10 fois plus longtemps. Cela montre que même pour les images de petite taille, le GPU offre des performances nettement supérieures. Pour l'image *Paris.ppm*, le GPU a pris en moyenne 135 µs pour le calcul de l'histogramme. Le CPU a pris en moyenne 2 438 µs, soit environ 18 fois plus longtemps. Pour l'image *Nuit.ppm*, le GPU a pris en moyenne 158 µs pour le calcul de l'histogramme. Le CPU a pris en moyenne 7 315 µs, soit environ 46 fois plus longtemps. L'écart de performance continue d'augmenter avec la taille de l'image. Pour l'image *MonSalon.ppm*, le GPU a pris en moyenne 448 µs pour le calcul de l'histogramme. Le CPU a pris en moyenne 43 768 µs, soit environ 98 fois plus longtemps. C'est l'écart de performance le plus important observé.

Ces résultats s'expliquent par le fait que le GPU est capable d'exécuter de nombreux threads en parallèle, ce qui permet de traiter chaque pixel de l'image simultanément. En revanche, le CPU traite les pixels de manière séquentielle, ce qui est beaucoup plus lent pour les grandes images.

Exercice 3

Voici les résultats obtenus de l'implémentation avec des images de différentes tailles et en mesurant le temps d'exécution du calcul de la fonction de répartition :

Taille des Images	Moyenne du temps d'exécution du calcul de la fonction de répartition sur GPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	82
Paris.ppm (562 000 pixels)	84
Nuit.ppm (1 764 000 pixels)	101
MonSalon.ppm (10 036 224 pixels)	184

En ce qui concerne le code, La fonction *run_Repartition* prend en entrée un histogramme des valeurs de l'image et un tableau pour stocker la fonction de répartition résultante. Elle utilise la fonction *inclusiveScan* pour calculer la fonction de répartition, qui est essentiellement la somme cumulative des valeurs de l'histogramme.

En ce qui concerne les résultats, On observe que le temps d'exécution du calcul de la fonction de répartition est très rapide sur le GPU. Par exemple, pour l'image *MonSalon.ppm* de taille 10 036 224 pixels, le temps d'exécution moyen sur le GPU est de seulement 184 µs.

Exercice 4

Voici les résultats obtenus en appliquant la transformation finale $T(x_i)$:

Figure : Image Hopper.railroad.ppm avant l'application de la transformation finale

Figure : Image Hopper.railroad.ppm après l'application de la transformation finale

Taille des Images	Moyenne du temps d'exécution du calcul de la transformation finale sur GPU (µs)	Moyenne du temps d'exécution du calcul de la transformation finale sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	84	308
Paris.ppm (562 000 pixels)	123	1 224
Nuit.ppm (1 764 000 pixels)	168	2 077
MonSalon.ppm (10 036 224 pixels)	757	16 109

En ce qui concerne le code, La fonction *run_Transformation* prend en entrée un tableau de valeurs (représentant les valeurs des pixels de l'image), un tableau représentant la fonction de répartition des valeurs, et un tableau pour stocker la transformation finale. Elle utilise *transformation_kernel*, pour calculer la transformation finale pour chaque valeur. Cette transformation est basée sur la fonction de répartition des valeurs et la taille totale de l'image.

En ce qui concerne les résultats, On observe que le temps d'exécution du calcul de la transformation finale est très rapide sur le GPU. Par exemple, pour l'image *MonSalon.ppm* de taille 10 036 224 pixels, le temps d'exécution moyen sur le GPU est de seulement 757 µs. En comparaison, le temps d'exécution moyen sur le CPU pour la même opération est de 16 109 µs, soit environ 21 fois plus long. Cela montre que le GPU est nettement plus rapide que le CPU pour le calcul de la transformation finale, et que cet écart de performance augmente avec la taille de l'image.

Il est important de noter que l'utilisation du GPU pour le calcul de la transformation finale des valeurs d'une image permet d'obtenir des performances très élevées. Cependant, il est important de noter que ces performances dépendent du contexte de l'utilisation et du matériel utilise.

Exercice 5

Voici les résultats obtenus en appliquant la version AHE décrite dans l'article :

Figure : Image Roy_Lichtenstein_Drowning_Girl.ppm avant l'application de la version AHE

Figure : Image Roy_Lichtenstein_Drowning_Girl.ppm après l'application de la version AHE avec Lambda = 1

Taille des Images	Moyenne du temps d'exécution de la version AHE sur GPU (µs)	Moyenne du temps d'exécution de la version AHE sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	98	420
Paris.ppm (562 000 pixels)	114	874
Nuit.ppm (1 764 000 pixels)	214	2 581
MonSalon.ppm (10 036 224 pixels)	838	18 999

Note : Dans les résultats du temps, nous utilisons lambda = 1. Faire varier lambda n'est pas significatif et n'affecte pas les performances.

En ce qui concerne le code, La fonction *run_Transformation* prend en entrée un tableau de valeurs (représentant les valeurs des pixels de l'image), un tableau représentant la fonction de répartition des valeurs, un tableau pour stocker la transformation finale, et une valeur lambda pour l'équation (6) de l'article. Elle utilise, *ahe_transformation_kernel*, pour calculer la transformation finale pour chaque valeur. Cette transformation est basée sur la fonction de répartition des valeurs, la taille totale de l'image, et la valeur lambda. L'équation (6) est appliquée à la volée lors de la transformation finale de la valeur V de chaque pixel.

En ce qui concerne les résultats, On observe que le temps d'exécution du calcul de la transformation finale est très rapide sur le GPU. Par exemple, pour l'image MonSalon.ppm de taille 10 036 224 pixels, le temps d'exécution moyen sur le GPU est de seulement 838 µs. En comparaison, le temps d'exécution moyen sur le CPU pour la même opération est de 18 999 µs, soit environ 23 fois plus long.

En ce qui concerne l'implémentation, il y a eu plusieurs raisons à cela :

1- Application de l'équation (6) à la volée :

L'équation (6) est appliquée à la volée lors de la transformation finale de la valeur V de chaque pixel. Cela permet d'éviter les erreurs numériques importantes qui pourraient se produire si l'équation était directement appliquée avant le calcul de la fonction de répartition en utilisant des entiers.

2- Utilisation d'un foncteur pour la conversion des valeurs :

Un foncteur est utilisé pour convertir les valeurs en float en valeurs entières non signées. Cela permet d'obtenir des valeurs discrètes pour l'histogramme et la fonction de répartition, ce qui est nécessaire pour le calcul de la transformation finale.

3- Choix de la valeur lambda :

La valeur lambda dans l'équation (6) peut être ajustée pour contrôler le niveau de contraste de l'image transformée. Une valeur de lambda égale à 0 donne l'algorithme HE (Histogram Equalization), tandis que des valeurs plus élevées donnent un contraste plus élevé.

Exercice 6

Dans cet exercice j'ai essayé de le faire mais malheureusement je ne comprends pas trop pourquoi ça ne marche pas, j'ai un tout petit plus de contraste par rapport à l'image attendu, voici les images

Figure : Image Nuit_WHE.ppm de référence après l'application de la version WHE

Figure : Image Nuit_WHE.ppm (que j'obtiens avec mon algo) après l'application de la version WHE

- Voici les explications de chaque kernel et des calculs :
  - buildHistogramAndVarianceSum_kernel :

    Cette fonction est conçue pour construire un histogramme des intensités de pixel et calculer la somme des variances pour une image. Elle opère sur une image représentée sous forme d'un tableau 1D de valeurs en flottant (dev_inputValue), où chaque valeur représente l'intensité d'un pixel. Voici une explication détaillée de la fonction :

    - Initialisation de la mémoire partagée :

      La fonction initialise d'abord les variables de mémoire partagée. Ces variables sont partagées entre tous les threads au sein d'un bloc. shared_sum et shared_count sont utilisées pour calculer l'intensité moyenne des pixels. shared_histo est un tableau utilisé pour construire l'histogramme des intensités de pixel. shared_sum_squared_diff est utilisée pour calculer la variance.
    - Calcul de l'indice de thread :

      Chaque thread obtient un indice unique (tid), qui est calculé en fonction de l'indice de bloc (blockIdx.x), du nombre de threads par bloc (blockDim.x), et de l'indice de thread dans le bloc (threadIdx.x). Le pas (stride) est le nombre total de threads lancés par le noyau.
    - Construction de l'histogramme et calcul de la variance :

      Chaque thread traite plusieurs pixels de l'image. Pour chaque pixel, il incrémente le bin correspondant dans l'histogramme (shared_histo[xi]), et calcule la variance locale en fonction de la différence entre l'intensité du pixel et les intensités de ses voisins de gauche et de droite. La variance est ensuite ajoutée à dev_weight[xi].

- Calcul de la moyenne :

  Après que tous les threads ont fini de traiter les pixels, ils se synchronisent avec __syncthreads(). Ensuite, le premier thread de chaque bloc calcule l'intensité moyenne des pixels et la stocke dans dev_weight[256].
- Calcul de la variance :

  Si le nombre de pixels traités est supérieur à 0, la fonction calcule la variance comme la moyenne des différences au carré par rapport à l'intensité moyenne des pixels. Cela est stocké dans dev_weight[258].
- Copie de l'histogramme :

  Enfin, la fonction copie l'histogramme partagé dans la mémoire globale (dev_histo).

- buildCumulativeDistributionFunction_kernel :

  Cette fonction calcule la fonction de distribution cumulative (CDF) des intensités de pixel dans l'image. Tout d'abord, elle calcule l'ID du thread (tid), qui est unique pour chaque thread. Si tid est inférieur à 256 (la plage des intensités de pixel dans une image 8 bits), elle calcule la somme des poids pour toutes les intensités jusqu'à tid. Cela revient essentiellement à construire la CDF. Ensuite, elle normalise cette somme en la divisant par la somme totale des poids (plus un terme de régularisation lambda pour éviter la division par zéro). Le résultat est stocké dans dev_cdf[tid].

- applyTransformation_kernel :

  Cette fonction applique une transformation aux intensités de pixel en fonction de la CDF. Tout d'abord, elle calcule l'ID du thread (tid), qui est unique pour chaque thread. Si tid est inférieur à size (le nombre total de pixels dans l'image), elle récupère l'intensité du pixel à tid et la convertit en unsigned char (xi). Ensuite, elle récupère la valeur de la CDF pour cette intensité (dev_cdf[xi]) et la normalise en la divisant par la valeur de CDF maximale (dev_cdf[255]).Le résultat est multiplié par 255 (la valeur d'intensité maximale pour une image 8 bits) et stocké dans dev_outputValue[tid]. Cela mappe efficacement l'intensité d'origine sur une nouvelle intensité basée sur la CDF, ce qui peut permet d'égaliser l'histogramme de l'image.

Le problème viendrait de la fonction buildHistogramAndVarianceSum_kernel,plus précisément dans le calcul de la variance local, mais malheureusement je n'ai pas réussi à le trouver.