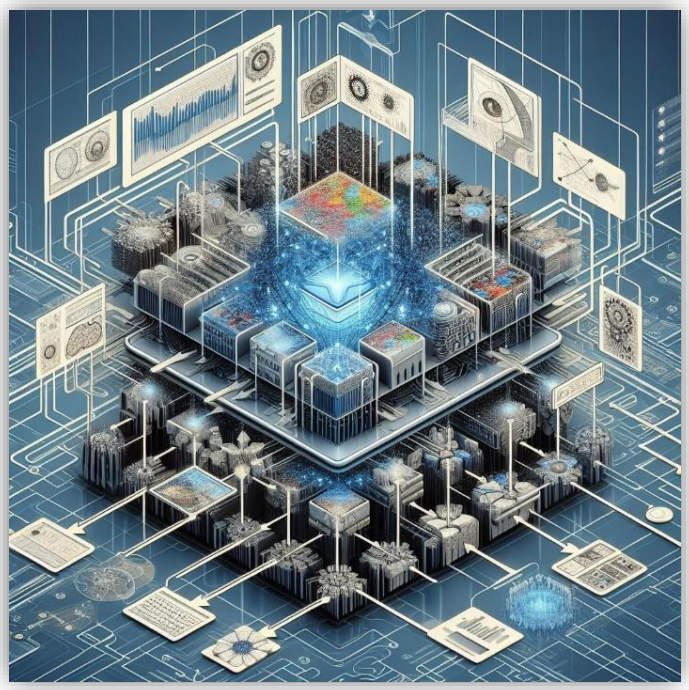

Sommaire

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

Exercice 1

Voici les résultats obtenus en appliquant le MAP unaire :

Figure : Image Nuit.ppm Avant l'application du filtre SEPIA

Figure : Image Nuit_sepia.ppm Après l'application du filtre SEPIA

Taille des Images	Moyenne du temps d'exécution du patron MAP Unaire sur GPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	105
Paris.ppm (562 000 pixels)	159
Nuit.ppm (1 764 000 pixels)	222
MonSalon.ppm (10 036 224 pixels)	875

En ce qui concerne le code qui met en œuvre la fonction UnaryMap. La fonction UnaryMap applique une fonction spécifique (le foncteur) à chaque élément d'un tableau source (src) et stocke le résultat dans un tableau de destination (dst). La fonction UnaryMap détermine d'abord la taille optimale du bloc pour le noyau unary_map en utilisant cudaOccupancyMaxPotentialBlockSize. Ensuite, on calcule le nombre de blocs nécessaires pour couvrir tout le tableau. Le noyau unary_map est ensuite lancé avec cette configuration de grille et de bloc. Dans le noyau unary_map, on applique le foncteur à chaque élément du tableau source. On utilise un index global calculé à partir de l'index du thread et de l'index du bloc. Un garde-fou est utilisé pour s'assurer que l'index est dans les limites du tableau, ce qui est crucial car la taille du tableau peut ne pas être un multiple de la taille du bloc.

En ce qui concerne les résultats, ils montrent le temps d'exécution moyen de la fonction UnaryMap sur différentes images. Comme prévu, le temps d'exécution augmente avec la taille de l'image. Par exemple, pour l'image CaravaggioUrsula.ppm avec 289 800 pixels, le temps d'exécution est de 105 µs, tandis que pour l'image MonSalon.ppm avec 10 036 224 pixels, il est de 875 µs. Cela montre que l'implémentation est efficace et capable de traiter des images de grande taille de manière assez rapide. Le plus intéressant est de comparer ses résultats avec une version CPU afin de voir l'efficacité réelle de l'algorithme dans la version GPU.

Exercice 2

Voici les résultats obtenus en appliquant le MAP binaire :

Figure : Image CaravaggioUrsula.ppm Avant l'application du Thumbnail

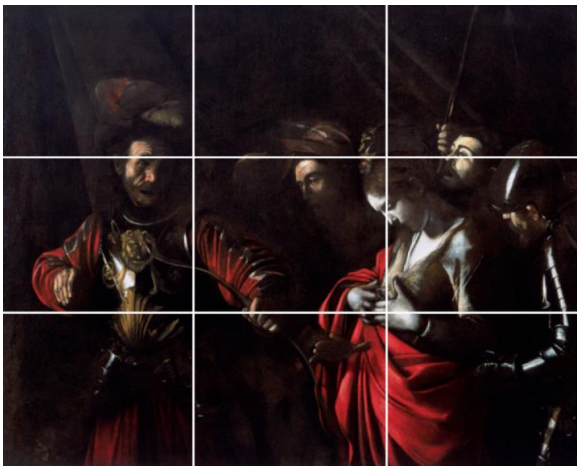

Figure : Image CaravaggioUrsula _thumbnail.ppm Après l'application du Thumbnail

Taille des Images	Moyenne du temps d'exécution du patron MAP Binaire sur GPU (µs)	Moyenne du temps d'exécution du patron MAP Binaire sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	92	109
Paris.ppm (562 000 pixels)	126	409
Nuit.ppm (1 764 000 pixels)	248	1 182
MonSalon.ppm (10 036 224 pixels)	1 105	6 222

En ce qui concerne le code qui est mis en place, la fonction BinaryMap applique une fonction spécifique (le foncteur) à chaque paire d'éléments correspondants de deux tableaux source (dev_a et dev_b) et stocke le résultat dans un tableau de destination (dev_result). Dans la fonction BinaryMap, on détermine d'abord la taille optimale du bloc pour le noyau binary_map en utilisant cudaOccupancyMaxPotentialBlockSize. Ensuite, on calcule le nombre de blocs nécessaires pour couvrir tout le tableau. Le noyau binary_map est ensuite lancé avec cette configuration de grille et de bloc. Dans le noyau binary_map, on applique le foncteur à chaque paire d'éléments correspondants des tableaux source. On utilise un index global calculé à partir de l'index du thread et de l'index du bloc. Un garde-fou est utilisé pour s'assurer que l'index est dans les limites du tableau, ce qui est crucial car la taille du tableau peut ne pas être un multiple de la taille du bloc.

En ce qui concerne les résultats, ils montrent le temps d'exécution à la fois sur GPU et CPU. Comme prévu, le temps d'exécution augmente avec la taille de l'image. On observe que le temps d'exécution pour des petites tailles est ~10 fois plus rapide sur GPU que sur CPU, alors qu'avec des grandes tailles on atteint un temps d'exécution qui est plus de ~100 fois plus rapide sur GPU que sur CPU. Par exemple, pour l'image CaravaggioUrsula.ppm avec 289 800 pixels, le temps d'exécution est de 92 µs sur le GPU contre 109 µs sur le CPU. Cela nous montre que l'implémentation est efficace et tire pleinement parti de la puissance de calcul parallèle du GPU contrairement avec la fonction MAP de la librairie standard C++ utilisé pour le TP3.

Exercice 3 & 4

Voici les résultats obtenus en appliquant le Gather et Scatter:

Figure : Image Paris.ppm Avant l'application du Scatter/Gather

Figure : Image Paris_scatter.ppm Après l'application du Scatter

Figure : Image Paris_gather.ppm Après l'application du Gather

Fondamentalement, dans les résultats des deux patrons, il n'y a pas de différence. Les deux permettront de faire une permutation aléatoire des blocs d'images, d'où l'avoir mis dans la même question. La différence réside dans la façon d'implémenter le patron et ses performances, C'est ce que nous allons explorer dans les prochaine parties.

→ Le Gather :

Taille des Images	Moyenne du temps d'exécution du patron Gather sur GPU (µs)	Moyenne du temps d'exécution du patron Gather sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	89	615
Paris.ppm (562 000 pixels)	148	962
Nuit.ppm (1 764 000 pixels)	246	3 362
MonSalon.ppm (10 036 224 pixels)	1 267	12 493

→ Le Scatter :

Taille des Images	Moyenne du temps d'exécution du patron Scatter sur GPU (µs)	Moyenne du temps d'exécution du patron Scatter sur CPU (µs)
CaravaggioUrsula.ppm (289 800 pixels)	98	714
Paris.ppm (562 000 pixels)	128	1 123
Nuit.ppm (1 764 000 pixels)	200	4 759
MonSalon.ppm (10 036 224 pixels)	894	15 735

En ce qui concerne le code fourni qui implémente les opérations de Gather et Scatter pour l'exécution sur un GPU. Ces opérations appliquent une fonction spécifique (le foncteur) à chaque élément d'un tableau source et stockent le résultat dans un tableau de destination. Les indices sont calculés en fonction de l'index du bloc et de l'index du thread dans le bloc. Un garde-fou est utilisé pour s'assurer que l'index est dans les limites du tableau.

En ce qui concerne les résultats, celles-ci montrent le temps d'exécution moyen de ces opérations sur différentes images, à la fois sur GPU et CPU. Comme prévu, le temps d'exécution augmente avec la taille de l'image. De plus, l'exécution sur le GPU est nettement plus rapide que sur le CPU. On peut observer un facteur de différence significatif entre les temps d'exécution sur le CPU et le GPU. Ce facteur de différence est calculé en divisant le temps d'exécution sur le CPU par le temps d'exécution sur le GPU.

Analyse des résultats pour l'opération Gather :

Pour l'image CaravaggioUrsula.ppm de 289 800 pixels, le temps d'exécution sur le GPU est de 89 µs, tandis que sur le CPU, il est de 615 µs. Cela signifie que l'opération Gather est environ 6,9 fois plus rapide sur le GPU que sur le CPU pour cette image.

Pour l'image MonSalon.ppm de 10 036 224 pixels, le temps d'exécution sur le GPU est de 1 267 µs, tandis que sur le CPU, il est de 12 493 µs. Cela signifie que l'opération Gather est environ 9,9 fois plus rapide sur le GPU que sur le CPU pour cette image.

Ces résultats montrent que l'opération Gather bénéficie grandement de l'exécution sur le GPU, en particulier pour les images de grande taille.

Analyse des résultats pour l'opération Scatter :

Pour l'image CaravaggioUrsula.ppm de 289 800 pixels, le temps d'exécution sur le GPU est de 98 µs, tandis que sur le CPU, il est de 714 µs. Cela signifie que l'opération Scatter est environ 7,3 fois plus rapide sur le GPU que sur le CPU pour cette image.

Pour l'image MonSalon.ppm de 10 036 224 pixels, le temps d'exécution sur le GPU est de 894 µs, tandis que sur le CPU, il est de 15 735 µs. Cela signifie que l'opération Scatter est environ 17,6 fois plus rapide sur le GPU que sur le CPU pour cette image.

Ces résultats montrent que l'opération Scatter bénéficie également grandement de l'exécution sur le GPU, en particulier pour les images de grande taille.

Ces résultats démontrent clairement l'avantage de l'utilisation du GPU pour les opérations de Gather et Scatter. Le GPU, avec sa capacité à exécuter de nombreux threads en parallèle, est particulièrement adapté aux opérations qui peuvent être parallélisées, comme c'est le cas pour les opérations de Gather et Scatter sur des images. Cela est particulièrement vrai pour les images de grande taille, où le gain de performance du GPU par rapport au CPU est le plus notable. Il reste important de dire que les facteurs de différence peuvent varier en fonction de la taille de l'image et de la complexité de l'opération effectuée. De plus, ils dépendent également des spécificités du matériel utilisé, à savoir le modèle du CPU et du GPU.

Exercice 5

Voici les résultats obtenus en appliquant le Filtre :

Figure : Image Raffael_012.ppm Avant l'application du Filtre

Figure : Image Raffael_012_filtered.ppm Après l'application du Filtre avec une taille de 8

Schémas de répartition des blocs	Moyenne du temps d'exécution sur l'image Raffael_012_filtered.ppm (µs)
(1, 1 024)	480
(8, 128)	3 110
(16, 64)	4 777
(32, 32)	4 680
(64, 16)	4 801
(128, 8)	2 478
(1 024, 1)	399

En ce qui concerne le code fourni, c'est une implémentation d'une opération de filtrage sur une image. Le filtre est appliqué à chaque pixel de l'image en utilisant un noyau, qui est lancé avec une configuration spécifique de blocs et de threads.

Dans le noyau kernelFilter, chaque thread calcule un index global à partir de son index de bloc et de thread, et applique le filtre si cet index est dans les limites de l'image. Le filtre est appliqué en sommant les produits des pixels voisins et des coefficients du filtre, et le résultat est écrit dans l'image de sortie.

La fonction run_filter prépare les données et lance le noyau. Elle calcule le nombre de blocs nécessaires pour couvrir tous les pixels de l'image, lance le noyau avec cette configuration, et synchronise le périphérique pour s'assurer que tous les threads ont terminé avant de continuer.

En ce qui concerne les résultats, On observe que le temps d'exécution est le plus court lorsque le nombre de blocs est le plus élevé et le nombre de threads par bloc est le plus faible, c'est-à-dire pour la configuration (1 024, 1). Cela peut potentiellement dire que cette configuration permet une meilleure

utilisation des ressources du GPU, probablement en raison d'une meilleure répartition de la charge de travail entre les threads.

Inversement, le temps d'exécution est le plus long pour les configurations où le nombre de blocs est faible et le nombre de threads par bloc est élevé, comme (8, 128) et (16, 64). Cela pourrait être dû à une sous-utilisation des ressources du GPU, car un nombre plus faible de blocs signifie moins de parallélisme au niveau des blocs. Cependant, il est important de noter que la configuration optimale peut dépendre de nombreux facteurs, y compris les spécificités du matériel GPU utilisé et la nature de l'opération effectuée.

Finalement, il est important de noter que l'accès coalescent à la mémoire peut avoir un impact significatif sur les performances. L'accès coalescent se produit lorsque les threads d'un même warp accèdent à des adresses de mémoire consécutives, ce qui permet d'optimiser l'utilisation de la bande passante mémoire. Dans le code, l'accès à la mémoire est effectué de manière coalescente, car les threads d'un même warp accèdent à des pixels voisins dans l'image source. Cela donc contribue probablement à l'efficacité de mon implémentation.