

***Sommaire***

[Exercice 1 2](#_Toc160484068)

[Exercice 2 3](#_Toc160484069)

[Exercice 3 & 4 4](#_Toc160484070)

[Exercice 4 5](#_Toc160484071)

[Exercice 5 6](#_Toc160484072)

|  |  |
| --- | --- |
| 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

**Une image contenant nuage, paysage, plein air, ciel

Description générée automatiquementUne image contenant nuage, paysage, plein air, eau

Description générée automatiquementVoici les résultats obtenus en appliquant le MAP unaire :**

Figure : Image Nuit\_sepia.ppm Après l'application du filtre SEPIA

Figure : Image Nuit.ppm Avant 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 suggère 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 :**

Une image contenant capture d’écran, Visage humain, art

Description générée automatiquementUne image contenant peinture, Visage humain, art, personne

Description générée automatiquement

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

Figure : Image CaravaggioUrsula.ppm Avant 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

Une image contenant ciel, plein air, bâtiment, nuage

Description générée automatiquement**Une image contenant ciel, plein air, nuage, éclairage

Description générée automatiquementUne image contenant ciel, plein air, silhouette, Rétroéclairage

Description générée automatiquementVoici les résultats obte** **nus 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

# Exercice 5

Dire si ça marche , faire une