Rapport projet CUDA

Membres du groupe:

THIAM Fatou
ISHIMWE Elis
GULLE Semih

Explication des filtres utilisées:

Filtre flou à boîte simple :

1. Le filtre flou à boîte simple, également connu sous le nom de filtre de moyenne, est un filtre de lissage utilisé pour réduire les détails et les variations d'une image. Il fonctionne en remplaçant chaque pixel de l'image par la moyenne des pixels voisins, pondérés de manière égale. Le résultat est une image plus douce et moins détaillée, ce qui peut être utile pour réduire le bruit ou les imperfections mineures.

Filtre gaussien :

2. Le filtre gaussien est un autre filtre de lissage utilisé pour flouter une image. Il est basé sur la fonction gaussienne, qui est une courbe en forme de cloche. Ce filtre attribue des poids plus importants aux pixels voisins les plus proches du pixel en cours de traitement, et des poids moins importants aux pixels plus éloignés. Cela crée un effet de flou plus naturel et progressif par rapport au filtre flou à boîte simple.

Filtre de Sobel Operator :

3. Le filtre de Sobel est un filtre de détection de contours largement utilisé dans le domaine du traitement d'images. Il est utilisé pour mettre en évidence les contours d'une image en calculant les gradients de luminosité dans les directions horizontale et verticale. Le filtre de Sobel est composé de deux masques, un pour chaque direction, qui sont appliqués à l'image originale pour obtenir les approximations des dérivées de l'image. En combinant ces deux résultats, on obtient une estimation de la magnitude du gradient et donc des contours de l'image.

Filtre laplacien de Gaussien :

4. Le filtre laplacien de Gaussien est utilisé pour accentuer les détails et les contours d'une image. Il combine deux opérations : la convolution avec un filtre gaussien pour flouter l'image, suivi de la convolution avec un filtre laplacien pour détecter les variations rapides de luminosité. Le filtre laplacien renforce les variations locales d'intensité, ce qui permet de mettre en évidence les contours et les détails fins.

Analyses des difficultés rencontrées lors de l’implémentation:

1. Complexité du calcul : Certains filtres, tels que le filtre gaussien et le filtre laplacien de Gaussien, impliquent des opérations de convolution qui peuvent être relativement coûteuses en termes de calcul. Pour les images de grande taille ou les filtres de grande taille, le temps de calcul peut devenir significatif. Il peut être nécessaire d'explorer des techniques d'optimisation telles que l'utilisation d'approximations, de tables de convolution précalculées ou de méthodes de sous-échantillonnage pour réduire la complexité du calcul.

2. Gestion des bords de l'image : Lorsque les filtres sont appliqués à l'image, il faut prendre en compte la gestion des bords. Par exemple, lors de la convolution, les pixels voisins en dehors des limites de l'image peuvent être nécessaires pour calculer la sortie. Différentes approches peuvent être utilisées pour gérer les bords, comme le zéro-padding, le miroir-padding ou le wrapping. Le choix de la méthode de gestion des bords peut influencer les performances et les résultats des filtres.

3. Impact sur les performances : L'application de filtres peut avoir un impact significatif sur les performances, en particulier pour les images de haute résolution ou lorsqu'elles sont appliquées en temps réel. Il peut être nécessaire d'explorer des techniques d'optimisation telles que la parallélisation, l'utilisation d'accélération matérielle (par exemple, GPU) ou l'optimisation des algorithmes pour améliorer les performances et réduire les temps de traitement.

Ainsi, la complexité du calcul, la gestion des bords, le choix des paramètres, l'impact sur les performances et la qualité des résultats sont quelques-unes des difficultés courantes rencontrées lors de l'implémentation et de l'optimisation des filtres. La maîtrise de ces défis nécessite une compréhension approfondie des filtres, des techniques d'optimisation et une itération expérimentale pour obtenir les résultats souhaités.

Nous allons discuter des choix d’optimisation apporté à l’ensemble des sources de la forme XXXOpt.cu .

En premier lieu , détaillons les étapes et choix d’optimisation de la boite simple.

1. Fonction CUDA pour le flou boîte :
La fonction `cudaFlouBoite` est une fonction CUDA qui effectue le flou boîte sur une partie de l'image. Elle est exécutée de manière parallèle sur le GPU. Chaque thread de calcul correspond à un pixel de l'image et effectue le calcul du flou boîte pour ce pixel.

2. Fonction principale `FlouBoiteGPU` :
Cette fonction est responsable de la gestion des données et de l'appel des fonctions CUDA pour appliquer le flou boîte sur l'image. Elle utilise la mémoire du GPU pour stocker les données source et les données de destination.

3. Allocation de mémoire sur le GPU :
Avant d'effectuer le flou boîte, la fonction `FlouBoiteGPU` alloue de la mémoire sur le GPU à l'aide de la fonction `cudaMalloc`. Deux tableaux de données sont alloués : `donneesSrcDevice` pour les données source et `donneesDstDevice` pour les données de destination.

4. Copie des données source de l'hôte vers le GPU :
Les données source de l'image sont copiées de la mémoire de l'hôte (CPU) vers la mémoire du GPU à l'aide de la fonction `cudaMemcpy`.

5. Configuration des dimensions de bloc et de grille :
Les dimensions de bloc et de grille sont configurées en fonction de la taille de l'image et de la taille de bloc spécifiée en paramètres. Cela permet de diviser l'image en blocs et de les traiter en parallèle sur le GPU.

6. Boucle d'itération pour le flou boîte :
Le flou boîte est appliqué en utilisant une boucle d'itération. Pour chaque itération, la fonction `cudaFlouBoite` est appelée en utilisant les données source et de destination appropriées.

7. Synchronisation des streams :
Après chaque itération, la fonction `cudaStreamSynchronize` est utilisée pour synchroniser les streams sur le GPU. Cela garantit que les calculs sont terminés avant de passer à l'itération suivante.

8. Copie des données de destination du GPU vers l'hôte :
Une fois toutes les itérations terminées, les données de destination sont copiées de la mémoire du GPU vers la mémoire de l'hôte à l'aide de la fonction `cudaMemcpyAsync`.

Efforts d'optimisation mis en place :

- Utilisation de CUDA et du parallélisme GPU pour effectuer le flou boîte de manière parallèle, exploitant ainsi les capacités de calcul massivement parallèle du GPU.
- Utilisation de la mémoire du GPU pour stocker les données source et de destination, ce qui réduit les transferts de données entre le CPU et le GPU et améliore les performances.
- Utilisation de streams CUDA pour exécuter les calculs en parallèle et de manière asynchrone, ce qui permet de chevaucher les transferts de données et les calculs, réduisant ainsi le temps total d'exécution.
- Utilisation de la mémoire partagée sur le GPU pour stocker les données temporaires nécessaires aux calculs du flou boîte, ce qui permet d'accélérer les accès mémoire et d'améliorer les performances.

Ces optimisations contribuent à accélérer le processus de flou boîte et à exploiter efficacement les capacités de calcul parallèle du GPU pour des performances optimales.

En second lieu, observons ce qui a été produit sur le filtre gaussien.

Ce code applique un flou gaussien sur une image en utilisant CUDA pour le calcul parallèle sur le GPU. Voici une explication du code :

1. Fonction de normalisation du noyau gaussien :
   La fonction `normaliserNoyauGaussien` est utilisée pour normaliser le noyau gaussien. Elle calcule la somme de tous les éléments du noyau, puis divise chaque élément par cette somme pour obtenir une somme totale de 1.

2. Fonction CUDA `applyGaussianBlur` :
   Cette fonction est exécutée en parallèle sur le GPU pour appliquer le flou gaussien sur l'image. Chaque thread de calcul correspond à un pixel de l'image. Les threads chargent le noyau gaussien normalisé dans la mémoire partagée et utilisent les éléments du noyau pour calculer le flou gaussien pour chaque pixel.

3. Fonction `flouGaussienGPU` :
   Cette fonction gère les opérations liées à CUDA, alloue de la mémoire sur le GPU, copie les données de l'hôte vers le GPU, configure les dimensions de bloc et de grille, et appelle la fonction CUDA `applyGaussianBlur` pour appliquer le flou gaussien.

4. Allocation de mémoire sur le GPU :
   La fonction `flouGaussienGPU` utilise `cudaMalloc` pour allouer de la mémoire sur le GPU pour les données source, les données de destination et le noyau gaussien normalisé.

5. Copie des données source de l'hôte vers le GPU :
   Les données source de l'image sont copiées de la mémoire de l'hôte (CPU) vers la mémoire du GPU à l'aide de `cudaMemcpy`.

6. Configuration des dimensions de bloc et de grille :
Les dimensions de bloc et de grille sont configurées en fonction de la taille de l'image et de la taille de bloc spécifiée en paramètres. Cela permet de diviser l'image en blocs et de les traiter en parallèle sur le GPU.

7. Boucle d'itération pour le flou gaussien :
Le flou gaussien est appliqué en utilisant une boucle d'itération. La fonction CUDA `applyGaussianBlur` est appelée à chaque itération pour effectuer le flou gaussien sur les données source et les stocker dans les données de destination. Les données source et de destination sont échangées à chaque itération pour les itérations précédentes, puis la dernière itération effectue le flou gaussien sur les données de destination finales.

8. Copie des données de destination du GPU vers l'hôte :
Une fois toutes les itérations terminées, les données de destination sont copiées de la mémoire du GPU vers la mémoire de l'hôte à l'aide de `cudaMemcpy`.

Ce code utilise CUDA pour exploiter le parallélisme massif du GPU et appliquer efficacement le flou gaussien sur l'image. Les opérations sont optimisées en utilisant la mémoire partagée sur le GPU pour stocker le noyau gaussien et en minimisant les transferts de données entre le CPU et le GPU.

Enfin , nous nous pencherons directement sur le filtre Laplacien de gauss en omettant le filtre sobel Operateur dont les choix d'optimisations sont résumé à travers les filtres déjà présenté .

Ce code applique le filtre du Laplacien de Gaussienne sur une image en utilisant CUDA pour le calcul parallèle sur le GPU. Voici une explication du code :

1. Fonction CUDA `LaplacianOfGaussianCUDA` :
Cette fonction est exécutée en parallèle sur le GPU pour appliquer le filtre du Laplacien de Gaussienne sur l'image. Chaque thread de calcul correspond à un pixel de l'image. Les threads utilisent un masque prédéfini de taille 5x5 pour effectuer le calcul du Laplacien de Gaussienne pour chaque pixel.

2. Fonction `LaplacienDeGaussienne` :
Cette fonction gère les opérations liées à CUDA, alloue de la mémoire sur le GPU, copie les données de l'hôte vers le GPU, configure les dimensions de bloc et de grille, et appelle la fonction CUDA `LaplacianOfGaussianCUDA` pour appliquer le filtre du Laplacien de Gaussienne.

3. Allocation de mémoire sur le GPU :
La fonction `LaplacienDeGaussienne` utilise `cudaMalloc` pour allouer de la mémoire sur le GPU pour les données source et les données de destination.

4. Copie des données source de l'hôte vers le GPU :

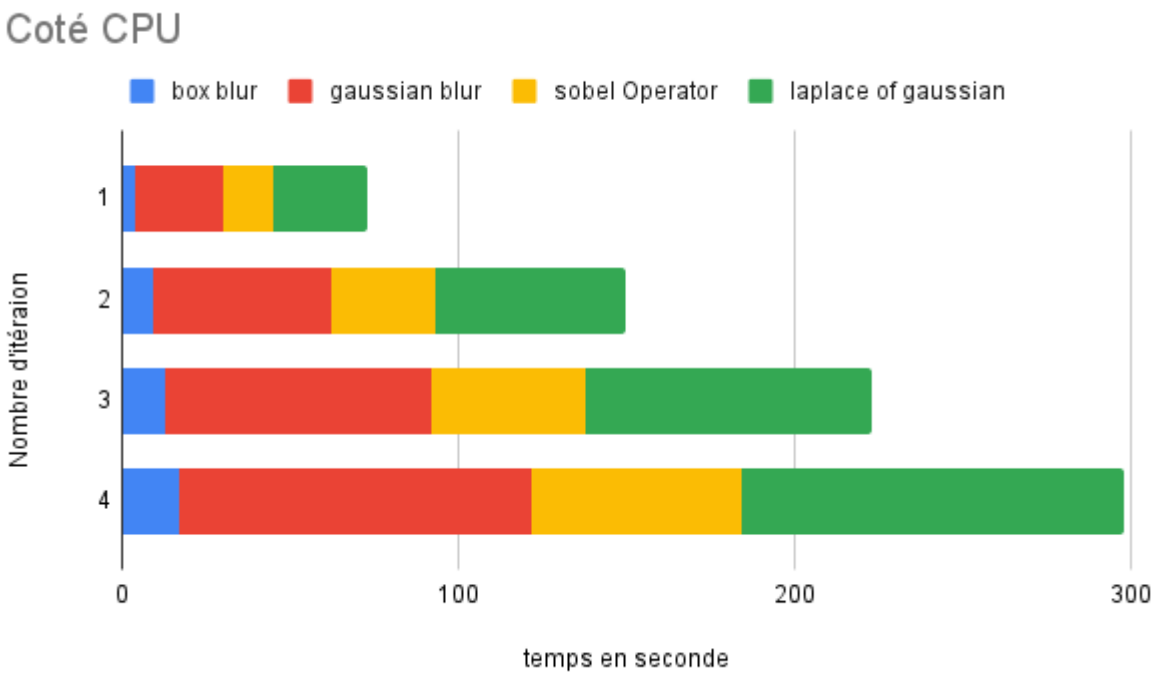
Les données source de l'image sont copiées de la mémoire de l'hôte (CPU) vers la mémoire du GPU à l'aide de `cudaMemcpy`.

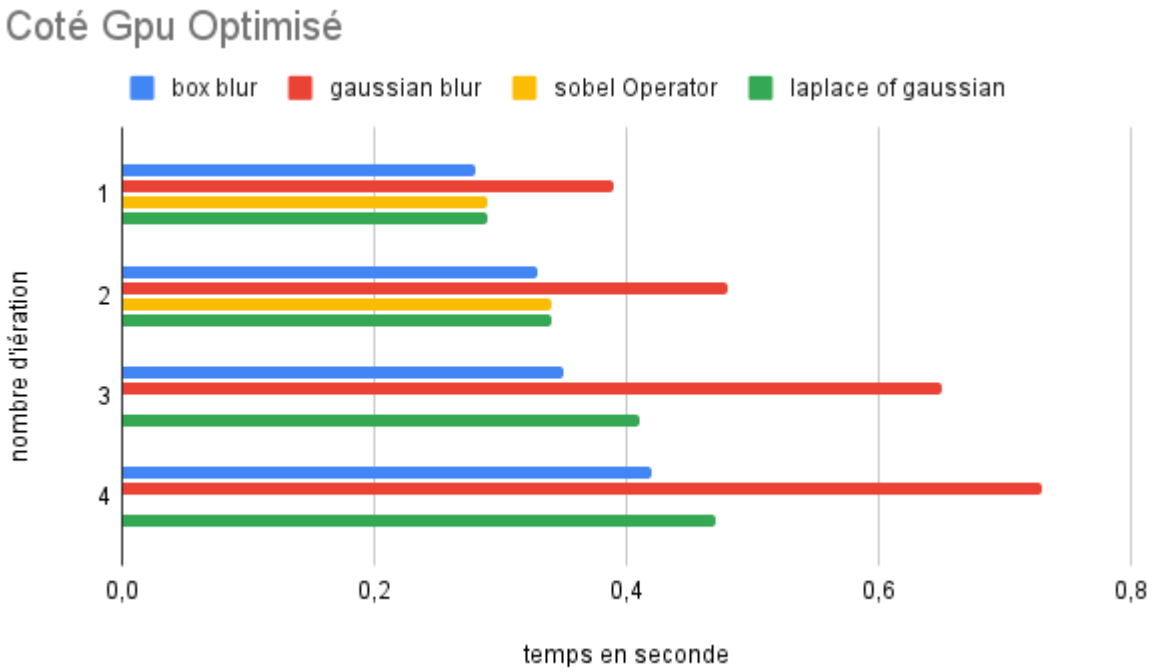
5. Configuration des dimensions de bloc et de grille :
Les dimensions de bloc et de grille sont configurées en fonction de la taille de l'image et de la taille de bloc spécifiée en paramètres. Cela permet de diviser l'image en blocs et de les traiter en parallèle sur le GPU.

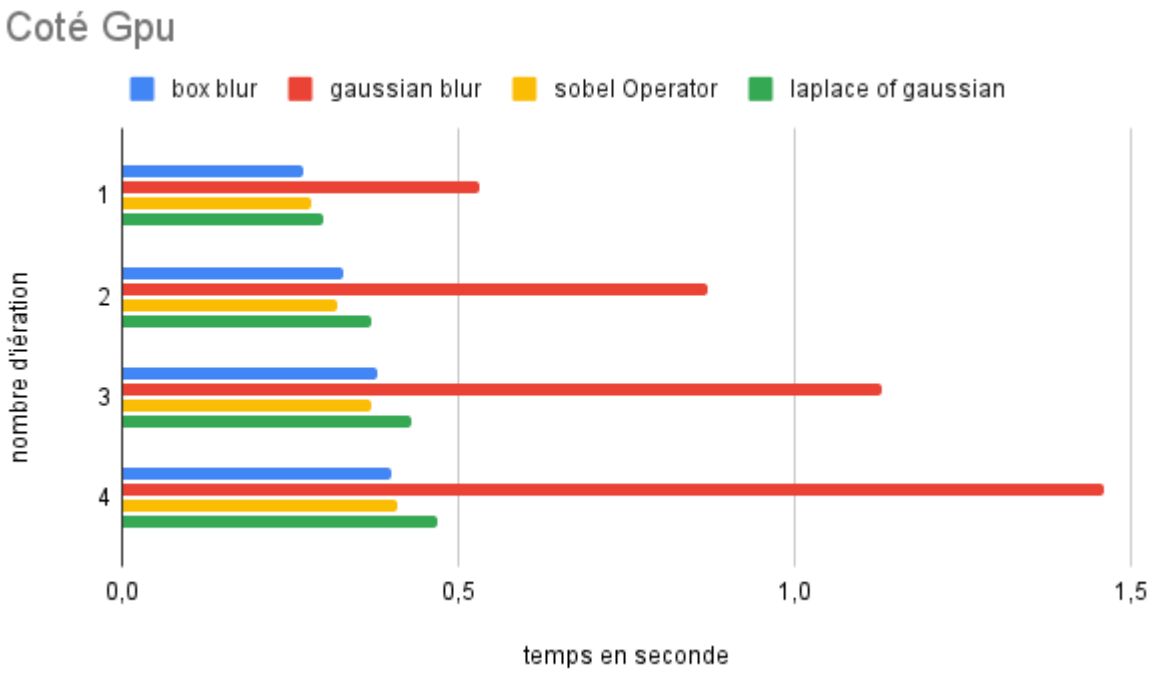
6. Boucle d'itération pour le filtre du Laplacien de Gaussienne :
Le filtre du Laplacien de Gaussienne est appliqué en utilisant une boucle d'itération. La fonction CUDA `LaplacianOfGaussianCUDA` est appelée à chaque itération pour effectuer le calcul sur les données source et les stocker dans les données de destination. Les données source et de destination sont échangées à chaque itération.

7. Copie des données de destination du GPU vers l'hôte :
Une fois toutes les itérations terminées, les données de destination sont copiées de la mémoire du GPU vers la mémoire de l'hôte à l'aide de `cudaMemcpy`.

Passons à la phase d'expérimentation :







Conclusion des expérimentations :

Le temps de calcul coté GPU et GPU optimisé est bien plus rapide comparé au CPU ce qui est cohérent.

Toutefois , même si le côté GPU optimisé parvient à limiter le temps de calculer notamment avec le filtre gaussian blur il est assez timide par rapport aux autres filtre.
Cette timidité peut être symbolisé par la combinaison de la mémoire partagé et des streams.

En effet, on a pu utilisé de façon malvenue les streams ce qui induit à des temps de calcul assez similaire .

En conclusion, la combinaison des streams et de la mémoire partagé est d'après nous une bonne idée mais encore faut il manier les streams correctement.
De plus définir un comparatif avec une optimisation utilisant uniquement la mémoire partagé aurait pu être intéressant par rapport au choix ou non de la combinaison avec les Stream .

Merci , de nous avoir lues et excusez nous pour le retard.